DATA SET VERSIONING THROUGH LINKED DATA MODELS

By

Benno Lee

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

Peter Fox, Thesis Advisor

Jim Hendler, Member

Deborah MacGuiness, Member

Beth Plale, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2018
(For Graduation August 2018)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Listings

ACKNOWLEDGMENT

I would like to thank my committee for their input into my research journey. I would like to thank Peter Fox for his guidance and advice. I would like to thank Kathy Fontaine for her guidance and time in editing this document. I would like to thank my lab mates for insightful and lengthy discussions.

ABSTRACT

Data sets invariably require versioning systems to manage changes due to an imperfect collection environment. Data versioning systems are employed to manage changes to data, logging new data sets and communicating that change to data consumers. Versioning discussion remains imprecise, lacking standardization or formal specifications. Many works tend to define versions based on previously established data project traditions rather than an analysis on the versioning characteristics of the data set. Provenance ontologies have begun to address some of the issues in capturing change information as linked-data, but remain incomplete. To improve completeness, a linked-data model must capture addition, invalidation, and modification (AIM) changes, connect versions to changing attributes, and maintain continuity between versions. Creating a version model enabled documentation which captures the differences between versions known as change logs to be exposed as linked-data, but the encoding process had an impact on the performance of the change log. From the behavior of provenance data which has similar properties to version data, the change log performance was expected to decrease by no more than 50%. Data versioning traditions have popularized the use of dot-decimal identifiers which indicate a categorical difference from the immediately prior version. Using the change counts from applying the data model, a more precise method of evaluating change distance was generated compared to the categorical differences enabled by dot-decimal identifiers. The greater precision enables changes to be assessed across time rather than across versions, improving the accuracy of a data set's change rate assessment.

A complete versioning model needed to satisfy three requirements. The requirement of capturing AIM changes means that all types of changes were captured by the model. Connecting versions to attributes means the changing parts of a version are appropriately attributed to the correct data object. Requiring that version continuity be preserved means that the version model can capture not just the interaction between two objects, but also a series of versions as commonly seen in

large data versioning systems. A provenance based model was only able to partially achieve version-attribute connections and version continuity. A log based model was then developed but could not satisfy version continuity. A third hybrid approach was created but did not meet the version to attribute connection requirement. A fully connected model was studied, but the model could not differentiate between AIM changes. The final model, instantiated as the Versioning Ontology (VersOn), separated the fully connected model into three forms to meet the AIM change capture requirement. VersOn explicitly connects version objects to the underlying attributes, fulfilling the version-attribute connectivity requirement. By propagating changes across attributes and maintaining relations between versions and changes in cases of missing attributes, our model maintains version continuity, meeting all requirements for completeness.

The instantiation of the versioning model enabled VersOn to be encoded into change logs using the Resource Description Framework in Attributes (RDFa) and JSON for Linked Data (JSON-LD). Adding more data into the change log had an impact on the performance, measured by content per storage size, but the reduction was expected to be no more than 50%. Scripts were used to automatically generate a change log standardized by VersOn, but the impact to performance exceeded expectations. The RDFa change log experienced a 90% reduction in performance while the JSON-LD change log experienced more than 95% reduction in performance. The increased storage utilization by the JSON-LD change log enabled less restricted implementation of VersOn into the change log.

Versioning graphs instantiated using VersOn separate changes into individual instances organized by type of change which can be counted to create a change distance. Dot-decimal identifiers the amount of change from the prior version into major, minor, or smaller categories. Using the change counts increases the precision at which change between versions were assessed. Extending the changes enabled the use of domain knowledge to additionally improve the precision of change assessment as appropriate to the domain. Through analysis of the Global Change Master Directory (GCMD) Keywords Version 8.5, we found that VersOn enables data consumers to assess data change in ways relevant to the consumer and independent of the producer's assessment of change as indicated by the dot-decimal identifier assigned to the version.

Change counts collected using VersOn were made across versions rather than across time. Using the version publication time, the change counts of GCMD Keywords were distributed across time. Some of the versions were discovered to bundle changes from much earlier than the publication of the previous version, meaning that the change rate using version publication time did not accurately reflect the change rate of the changes in bundles. Using the start time of the change rather than the version more accurately captures the change rate of the data. The data sets collected by the Earth Observing Laboratory (EOL) were found to have AIM change rates consistently different from the version publication rate. Since change counts measure the individual changes in the data set, the VersOn enabled change counts integrated with time provides a more accurate assessment of the data set change rate.

Our versioning model, implemented as VersOn, enables more complete change capture using linked-data and exposes the information stored in change logs as linked-data. The model achieves distinct AIM change capture, version-attribute connectivity, and version continuity. The impact to change log performance of including the VersOn entries into the log exceeds a 50% reduction because modifications were not summarized. The impact demonstrates the importance of being able to summarize change in generating manageable change logs. Our implementation of VersOn enables more precise assessment of change in comparison to dot-decimal identifiers by counting specific changes rather than using categorical assessments. Using VersOn, we were able to demonstrate methods to more accurately asses data set change rates by using the change time rather than the version publication time and using the VersOn change rate rather than the version publication rate. Through developing VersOn, we additionally found that the ontology enables data consumers to take more control over the change assessment process and become more independent of the version information defined the data producer. Analyzing datasets through VersOn also revealed that data sets have a tendancy towards a certain type of change as well as specific categories of versioning schedules.

GLOSSARY

addition	A version-to-attribute change indicating the attribute has been added to the version. xviii, 17, 18, 36, 40–43, 54, 58, 62, 67–71, 74, 75, 77, 81, 83–85, 87–89, 91–93
attribute	Part of a version used to determine if a version has changed. xvi, 37–44, 47, 48, 51, 57–59, 62, 64, 72, 73, 75, 90–92
change	A relationship between a version and an attribute or between two attributes indicating a difference between the entities. xvi, 17, 18, 40–42, 44, 48, 53–55, 57–59, 62, 67, 70, 71, 73, 74, 76, 77, 79, 80, 83–85, 87–89, 91, 93
change distance	A difference score between two objects composed of three components, each counting one of the AIM changes. 17, 18, 68, 74, 77, 80, 84, 86, 92
change log	A text document describing the changes between two versions. 11, 12, 16–18, 37, 38, 43, 44, 46–48, 51–59, 62, 64, 67, 68, 72, 75, 76, 85, 89, 92
invalidation	An attribute-to-version change indicating the attribute no longer appears in the version. xviii, 17, 18, 36, 41–43, 54, 58, 62, 67, 68, 70, 71, 74, 75, 81–85, 87–89, 91
linked-data	The collection of interrelated datasets on the Web of Data [1]. 2, 6, 11, 12, 16–18, 29, 39, 42, 51, 52, 55, 57, 58, 64, 66, 73–75, 91
modification	An attribute-to-attribute change indicating values of the attribute differ between versions. xviii, 17, 18, 36, 41, 42, 48, 51–54, 58, 59, 62, 64, 67, 68, 70, 71, 74, 75, 80, 81, 83, 85, 87–89

provenance	Information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness [2]. 1, 2, 9, 11, 37, 42
provenance distance	A measure of difference between two objects based on their provenance graphs. 32
version	An **expression** of a **work** which exists in comparison to another object and communicates the extent to which it diverges from that object as a result of provenance changes. xvi, xvii, 4, 10–13, 16–18, 37–44, 46, 47, 57–59, 64, 67, 68, 72–77, 79, 80, 83–93
versioning graph	Instantiation of a version transition or series of transitions using the Versioning Ontology. 18, 40, 45, 49, 51, 55, 57–59, 62, 64, 67, 68, 70, 72, 73, 75, 76, 92, 93
volatility	the likelihood of data change. 77, 80

ACRONYMS

AIM	addition, invalidation, and modification. xvi, 17, 18, 59, 76, 80, 83, 87, 93, 100
ARM	Atmospheric Radiation Measurement. 5
ATLAS	A Toroidal LHC ApparatuS. 23
CLPX	Cold Land Processes Field Experiment. 47
CVS	Concurrent Versions System. 22
DOI	Digital Object Identifier. 20, 21, 28
EOL	Earth Observing Laboratory. 79–81, 86, 88, 93
FRBR	Functional Requirements for Bibliographical Records. 3, 5, 40, 57
GAST	Global Alignment for Sequence Taxonomy. 67, 75
GCMD	Global Change Master Directory. 64, 66, 67, 73, 77, 84, 90, 93
HCLS	Health Care and Life Sciences. 5, 36
HTML	HyperText Markup Language. 29, 30, 48, 52, 53, 55, 57, 59
IFLA	International Federation of Library Associations and Institutions. 3
IQR	InterQuartile Range. 81
JSON	JavaScript Object Notation. xviii, 31, 48, 51
JSON-LD	JSON for Linked Data. 31, 48, 51, 52, 55–57, 64, 68, 89
KMS	Key Management Service. 66, 77
MBVL	Marine Biodiversity Virtual Laboratory. 64, 66, 76, 90, 92
NASA	National Aeronautics and Space Administration. 1, 11, 47, 66, 68
NCA	National Climate Assessment. 9
NCAR	National Center for Atmospheric Research. 79

OPM	Open Provenance Model. 7, 11
PAV	Provenance, Authorship, Versioning. 10, 11, 36
PROV-O	W3C Provenance Ontology. 2–4, 8, 9, 11, 36, 40, 73, 91
PURL	Persistent Uniform Resource Locator. 20
RCS	Revision Control System. 21, 25
RDF	Resource Description Framework. 66
RDFa	Resource Description Framework in Attributes. 29–31, 48, 51, 52, 55–57, 59, 62, 64, 89, 92
RDP	Ribosomal Database Project. 67, 71, 75
SKOS	Simple Knowledge Organization System. 66
SPINGO	Species-level IdentificatioN of metaGenOmic amplicons. 67, 75
SVM	Software Versioning Manager. 12, 22
URI	Uniform Resource Identifier. 28, 59, 62, 68, 74
URL	Uniform Resource Locator. 20, 62
UUID	Universally Unique Identifier. 68
VersOn	Versioning Ontology. 16–18, 58, 64, 85, 89–92, 99, 100
W3C	World Wide Web Consortium. xix, 2, 5, 8

CHAPTER 1
INTRODUCTION

1.1 Definitions of Version

Data sets invariably require versioning systems to manage changes due to an imperfect collection environment. Data versioning systems are employed to manage changes to data, logging new data sets and communicating that change to data consumers. Versioning discussion remains imprecise, lacking standardization or formal specifications. Many works tend to define versions around examples and local characteristics but lack a broader foundation. This work contributes to the discussion by capturing version relationships into a linked data model, taking inspiration from provenance models that incorporate versioning concepts such as PROV and Provenance, Authorship, Versioning (PAV) ontologies.

Using the term ‘version’ in the vernacular has become so pervasive that few documents formally define it. Barkstrom describes versions as **homogeneous groupings** used to control, “production volatility induced by changes in algorithms and coefficients as result of validation and reprocessing,” [3]. The **groupings** he mentions are a method of separating data objects such that they have similar scientific or technical properties. In order to determine when these properties have changed, he leverages the National Aeronautics and Space Administration (NASA) Earth Science workflow model shown in Figure 1.1. The model describes the formal stages of processing to turn a raw remote sensing signal from satellite instruments into global aggregate summaries [3]. The workflow model, therefore, describes a data object’s provenance, the information about how a piece of data or thing was created and used to determine its quality, reliability or trustworthiness [2], not the version differences. Provenance provides the mechanism to explain differences between **groupings** exposed by versioning activities. Understanding the workflow model reveals that changes to either the algorithms or parameter files will force a change in the resulting data, creating a new version of the output data. Each time the provenance changes, the levels take on a layer cake appearance as shown in Figure 1.2.

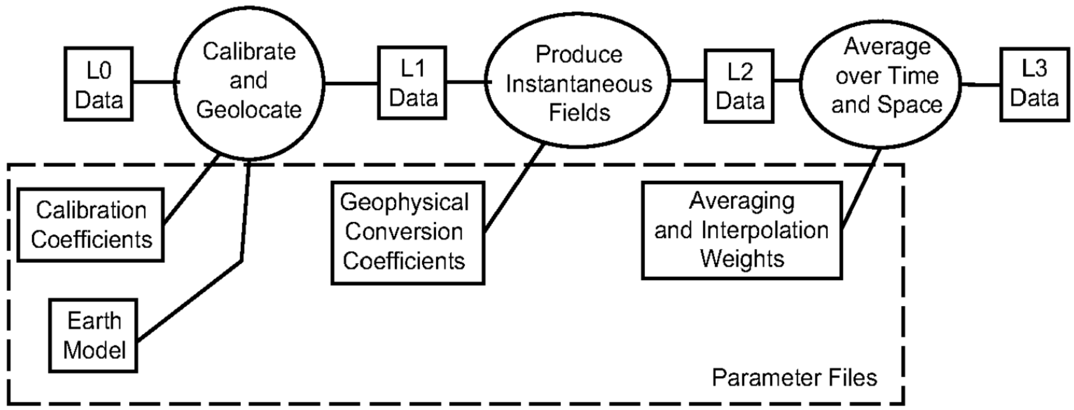

Figure 1.1: National Aeronautics Space Administration Earth Science organizes its data into three levels depending on how much the data has been aggregated and processed from the original sensor measurements. Figure 1 from [3]

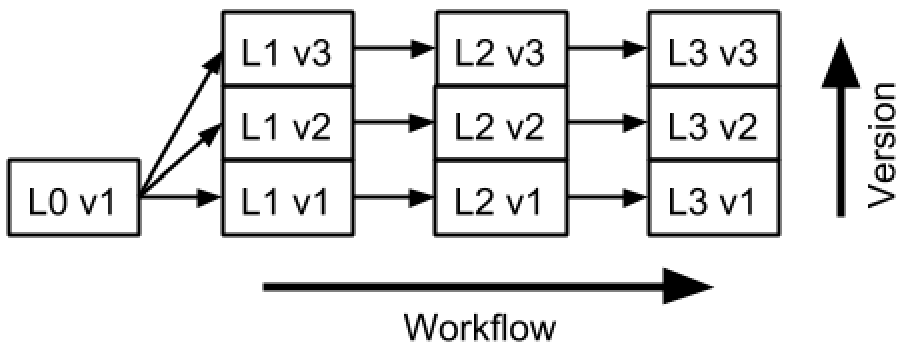

Figure 1.2: National Aeronautics and Space Administration Earth Science workflow model gains layers as changes to provenance induces version groupings to expand. The new layer is homogeneous with other layers in the level, forming a grouping. Level 0 cannot gain more layers because manipulating the source readings would invalidate the data.

Unlike provenance, version information describes relations not along the workflow, but in an orthogonal direction because change passes through the layers.

The conflation can also be seen in linked-data provenance models, Web of Data resources which describe the provenance relating data sets, where version concepts and links are included into the ontology. The W3C Provenance Ontology (PROV-O) is the World Wide Web Consortium (W3C) recommended method to document provenance, connecting versions using the property *prov:derivation*. The *prov:derivation*, covered more in Section 2.4.2, is defined as, "a transformation of an entity into another, an update of an entity resulting in a new one, or the construction of a new entity based on a pre-existing entity," [4]. The PROV-O definition establishes the existence of at least two distinct objects as a necessary condition for versioning. An additional result of the PROV-O definition is that an object cannot be a version of itself.

Another definition comes from Tagger at University College London on versioning of biology data in which versions are a, "semantically meaningful snapshot of a design object (an artifact) at a point in time," [5]. He, unfortunately, does not further clarify what he means by semantically meaningful since the requirements to be meaningful varies based on the application. He likely means that the design object must be complete or whole, interpretable in purpose by itself and not a subset. The design object is used in the context of database objects, referring to an incomplete final product, and unifies the versions as their primary subject, capturing the object's state over the course of its design.

The International Federation of Library Associations and Institutions (IFLA) formed a study group in February 1997 to, "produce a framework that would provide a clear, precisely stated, and commonly shared understanding of what it is that the bibliographic record aims to provide information about, and what it is that we expect the record to achieve in terms of answering user needs" [6]. A common problem in library sciences is the organization and documentation of multiple editions of a book or multiple translations of a book in a standardized manner. A primary result of the group meeting is a standardized vocabulary to discuss distinct iterations of a literary work which follows strong parallels in data versioning. The Functional Requirements for Bibliographical Records (FRBR) avoids the terms **edition** and **version** since "those terms are neither clearly defined nor uniformly applied" [6]. Instead, they use the terms: **work**, **expression**, and **manifestation**. A **work** refers to the abstract concept of a creative or artistic idea. **Expressions** are then different forms of that particular **work**, embodying the most similar term to the previous definitions of versions. A **manifestation** is the physical embodiment of an **expression**. These three terms and their hierarchy establish a repeating theme throughout other versioning works. The breakdown of terms identifies a need to differentiate between the granularity or scale of changes as differences in **works** are more significant than differences in **expressions**. Models focused on versioning rather than provenance will also have a hierarchical structure as covered in Section 2.2.

Combining these myriad of definitions, the Barkstrom definition provides the mechanism to detect and explain the source of a version. The Tagger and PROV-O definitions explain the purpose of a version to expose differences between distinct objects. The FRBR definition contextualizes a version as a more granular **expression** of a larger **work**. The working definition in this dissertation of a version is 'an **expression** of a **work** which exists in comparison to another object and communicates the extent to which it diverges from that object as a result of provenance changes'.

Although each definition disagrees on the form a version object takes on, all but PROV-O derivation agree that a version belongs to a larger collection of objects implementing a more abstract, ideal representation. Provenance provides the information necessary to explain "semantically meaningful" for the Tagger definition as *prov:Derivation* captures when a data object diverges into a new object.

1.2 Why Versioning is Important

> "If scientific data production were easy, instruments would have stable calibrations and validation activities would discover no need for corrections that vary with time. Unfortunately, validation invariably shows that instrument calibrations drift and that algorithms need a better physical basis." [3]

Anyone who has used an iPhone or owned a video game console understands the basics of versioning. Companies brand sequential devices to indicate improvements in performance or capabilities. Basic numerical sequencing has given rise to a plethora of versioning systems used widely across a landscape of software and data. At the very core, versioning systems are a means of communication. Data producers communicate to consumers how much the producer has changed the data. The producer can also communicate through time using logs and other documentation. In order to clearly communicate changes, ideas must be clearly formalized and defined and vice versa. Versioning systems help scientific workflows avoid losing work by managing transitions and changes while in operation [7]. Versioning systems provide necessary documentation which informs the transition to new methods and procedures [8]. Versioning systems provide accountability for the value of a project's data set when considering an agency's continued funding [9]. The natural evolution of versioning systems, however, have given rise to formal architecture operating on top of very informal concepts. Disagreements between the informal concepts propagate through the formal architecture, causing breakdowns in interoperability between heterogeneous systems, especially when users are not aware of biases in informal versioning concepts [10]. In this dissertation, we identify gaps in versioning practices which result from tradition and develop a data model to more completely capture the interactions involved in versioning.

1.2.1 Change Capture Completeness

Data provenance has achieved major adoption and implementation goals in data management discussion with the adoption of PROV-O [4] as a W3C recommendation and PROV-O's utilization in the National Climate Assessment (NCA) [11]. The inclusion of versioning concepts within the ontology illustrates the close relationship between provenance and change information. Provenance's prominence in the data management field defines a specific view of change capture. Because provenance focuses on capturing the entities and activities used to produce a data object, it does not completely capture the change differences between objects. Delving deeper and comparing the parts of the objects lies outside the scope of provenance information. In order to achieve completeness, a linked-data model must include the concepts of addition, invalidation, and modification. The model must relate the changing parts which differentiate objects as separate versions with the objects. The model must also maintain a continuity between versions to preserve the historical evolution of a data object.

1.2.2 Change Log Performance

A change log is a document explaining the differences between two versions [12]. Every change logged within the change log requires a certain amount of storage space to document. One of the ways to measure change log performance is to compute the number of entries in the change log relative to the space necessary to store the information. The amount of change information generated for an object can be related to provenance information since changes to activities and inputs result in new data objects which can be compared to the original object as a version, generating change information. Buneman finds that provenance data consumes storage space on the order of the original data, or after compression, the provenance takes up twenty percent of the original space [13]. Including more data in a different form is guaranteed to decrease performance with the increased space utilization, but the reduction can be bounded. Since the change log contains the differences between two versions, therefore containing at most all the entries in the data set, encoding the change data into the log will decrease performance by at most fifty percent. The storage space necessary to hold the original change log and the change data is double the original data set size, but the entries remain the same since the change log and change data entries describe the same change.

1.2.3 Change Distance Precision

Versioning practices tend to follow specific traditions [14]; one tradition is the use of dot-decimal identifiers to communicate change distance [5]. A dot-decimal identifier is a series of numbers joined together by a decimal point with the left-most number signifying a grouping by major features and the right-most number signifying a grouping over the most minor features. The amount of change between two versions is determined by finding the left-most interval at which the identifiers differ. Because the identifiers traditionally denote a version's location in a sequence, only one interval is changed at a time and is only incremented by one. The identifiers broadly categorize the change distance between two versions. Because the number of changes between two versions is one or more, a range of change distances can fall within a category. The range suggests that counting changes allows a more precise basis to report change distance than dot-decimal identifiers.

1.2.4 Accuracy of Data Set Change Rate

Looking at just the differences between versions may give a false impression of the data set's rate of change since the comparison to versions assumes that the versions are distributed evenly through time. From Git [15] and other versioning systems [16], we can see versions can be arbitrarily distributed through time. Variable publication rates help increase the stability of a project by controlling the data volatility, the likelihood of data to change, by aggregating changes before publishing a new version. The division of changes across versions suggests that re-introducing time to change distance will give a more accurate reading of a data set's rate of change than the version publication rate and more accurate than changes across versions.

1.3 Scope and Assumptions

The data sets used to develop the versioning model include static tabular databases, taxonomic trees, and collections of small data sets possessing around twelve to thirteen files on average. The static tabular databases contained on the order of thousands of records and the time between versions was on the order of months and years. A number of classes of data do not reflect the temporal and size characteristics of the utilized data sets including streaming data, which posses short intervals between versions, and big-data repositories, which posses orders of magnitude more records. Taxonomic trees do not represent many web ontologies because the ontologies may contain undirected cycles in the ontological structure. The versioning model can only be applied to data sets with more than one version since the version must be compared to another object.

The model assumes that multiple changes occur between each version change. The results in this thesis may not represent the performance of data sets which operate on single or small numbers of changes between each version of a data set. Transactional data sets like centralized data sets are not covered by the research work in this thesis. The model assumes that all newly appearing attributes have been added by the new version. All attributes which do not appear in the new version are assumed to be invalidated. The model assumes that every entry in a data set can be unique identified by an attribute or a combination of attributes. Data sets where changes cannot be associated with mutually exclusive groups cannot be correctly represented in the versioning model.

1.4 The Versioning Use Case

The research questions addressed by this thesis revolve around the check and retrieval stages, steps 3 and 4a, of the flow in the Versioning Use Case shown in Table 1.1. The goal of the Versioning Use Case is to allow the Producer to communicate changes of data tracked by the versioning system to the Consumer. The use case begins when a data producer wishes to distribute the next version of a data set. Note the use of next version instead of new version to signify that the Versioning Use Case also includes situations where the data is not new or describes concurrent events. The use case features the Producer and Consumer as actors using the versioning system as an intermediary. The actors do not follow separate use cases because each actor relies on the actions of the other actor to contextualize the actions in the use case.

The basic flow of the use case is seen in Figure 1.3 where the Producer logs a version into the versioning system, and the Consumer retrieves that data version. At a later point in time, the Producer can add any number of corrections to the versioning system. The repeated corrections reflect a practice of passively logging additional changes. At some final time after 0 or more corrections, the Consumer returns to check the versioning system for any changes to the data set the Consumer retrieved. If there are changes to parts pertinent to the Consumer, the Consumer retrieves the corrected data. In the interaction, the Producer only passively provides additional versions of the data while the responsibility of remaining up-to-date lies with the Consumer. The ability of the versioning system to communicate to the Consumer that the data has changed determines whether the use case succeeds or fails for the data consumer. The relationship creates a Producer/Consumer dynamic which influences the performance of the versioning system.

Table 1.1: Versioning Use Case Table

Use Case Name: New Version Publication & Retrieval
Goal: Record a new version of a data set and provide it to a data consumer.
Summary: The Producer creates a new version of their data and must record it to the Versioning System while providing the Consumer with the data
Actors: Producer, Consumer
Preconditions: Producer has supplied some data to the Versioning System. Consumer has retrieved the data from the Versioning System.
Triggers: Producer provides a different version of the data to the Versioning System.
Basic Flow: 1. Producer places the next version into the Versioning System. 2. Producer may repeat the previous action 0 or more times. 3. Consumer checks the data to see if there are changes. 4. If there are pertinent changes: (a) Consumer retrieves the updated data.
Alternate Flow: 1. Producer places the next version into the Versioning System. 2. Producer notifies Consumer that an alternate version is available. 3. Consumer retrieves the updated data.
Post Conditions: Producer has made the alternate version available. Consumer possesses the pertinent newly available version.

The alternate flow in Figure 1.4 requires additional information from the Consumer which is a means of notification. The flow begins the same as the basic flow, but after a single correction, the Producer notifies the Consumer that the data has changed. The notification causes the Consumer to come and retrieve the updated data set. The alternate flow poses a few problems in that the Producer must now manage notification information, but the Producer must also notify all consumers of the data which may cause scalability issues. Notice that in both flows, the Producer possesses the authority to determine change through data publication. The

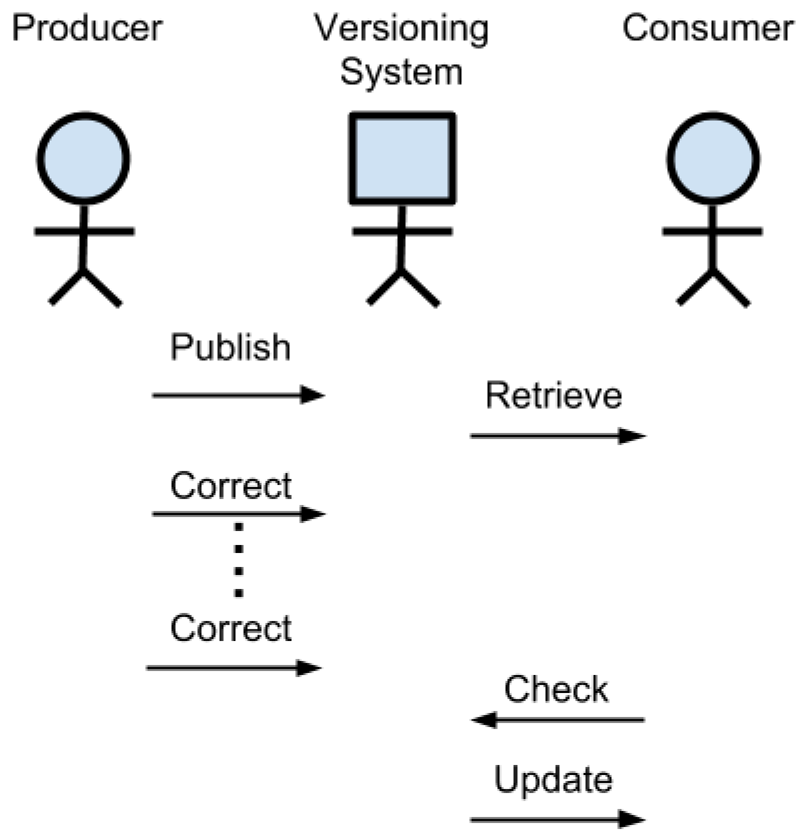

Figure 1.3: Basic Flow of the Versioning Use Case. The Producer adds a data set to be tracked by the versioning system. After the Consumer retrieves the data, the Producer makes a series of updates and corrections to the original data. The Consumer then returns to determine if there are changes to pertinent data files. The Consumer then updates as necessary.

Consumer only takes from the versioning system except in step three of the basic flow. It is only when the Consumer checks the versions in the system or is notified by the Producer that the Consumer can contextualize changes to the retrieved data set.

1.4.1 Research Question 1: What has changed?

The primary research question addressed in the following chapters pertains to step three in the basic flow of the use case. In order to execute step three, the Consumer must have a means of determining whether the data set is different from

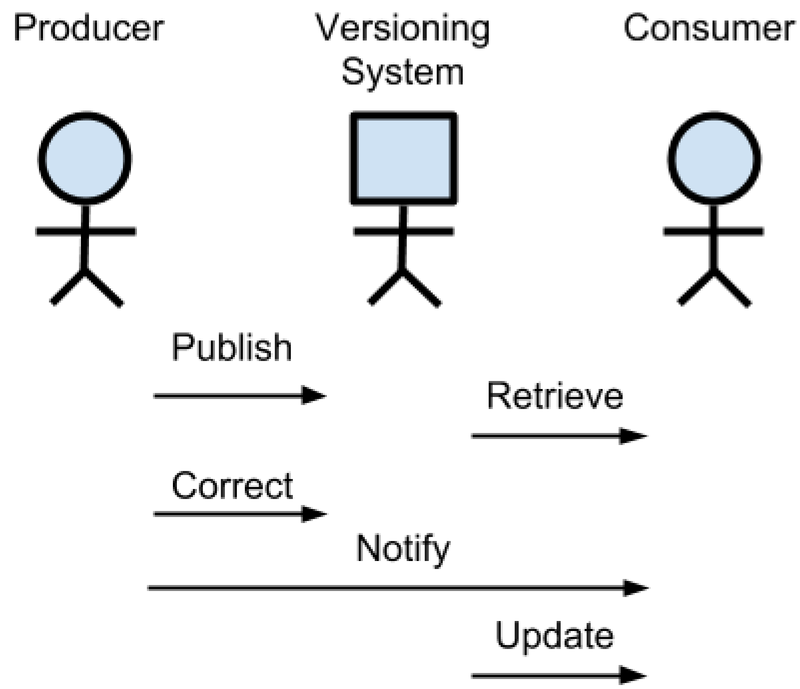

Figure 1.4: In the alternate flow, the authority to determine necessary updates lies with the Producer. After correcting data tracked by the versioning system, the Producer notifies the Consumer to update the Consumer's data.

the one the Consumer currently possesses. In a large number of software projects, especially open-source projects, change logs document for the Producer changes to the code base and communicate alterations to the Consumer. Very few data sets provide detailed change logs along with versions in the versioning system. Either general descriptions of changes are made or new documentation explaining usage is provided, requiring the data consumer to manually acclimate to changes.

The prevailing approach to addressing Research Question 1 is that automating the generation of change logs will allow the addition of linked-data into change log content. To encode changes as linked-data, I developed a versioning ontology called Versioning Ontology (VersOn). The additional data allows data consumers the ability to search, filter, and otherwise interact with change information. Different standards exist across various national agencies, but few fundamentals have been established on what needs to be included and how the information is to be captured or presented. I determine whether a linked-data versioning model would provide a standardized basis for discussing version change as well as allow tools to be developed to assist in step three for large data sets of which Consumers may only use a part.

1.4.2 Research Question 2: How much has changed?

A follow-up query to Research Question 1 is that once the changes have been determined, Consumers need to evaluate the impact of the changes. As previously mentioned, Producers sequentially label versions to communicate change, but many have adopted the practice of using decimals or dots to imply fractional changes, forming a dot-decimal identifier. The authority to determine the magnitude of change lies with the Producer since the label must be applied at publication. Version counts, defined in Chapter 5, provide a very bland description of the changes in a version by aggregating a range of changes under a sequential label. Using VersOn, the changes can be broken down into addition, invalidation, and modification (AIM) classifications which provide a quantitative method to determine change distance. I test whether the AIM change counts would tie change metrics back to data differences rather than to versions. I determine whether the AIM changes also provide a means to compare the difference between the amount of change declared by the Producer and quantity of change as seen by various Consumers.

1.4.3 Research Question 3: How fast does the data change?

Once a standardized metric for determining change distance has been determined, assessing change rate logically follows. Up to this point, evaluating the differences between versions has been discussed without respect to time. Notice from the Versioning Use Case that the Producer has the ability to control the rate of version release by determining when to push corrections into the versioning system, meaning the time between versions is not required to be consistent. Variable release times allows the Producer to control the data volatility, the likelihood of a data set to change, by aggregating changes over time. Without considering time, change distance across versions misrepresents the actual change rate of a data set. Looking at just the version publication rate also misrepresents the actual change rate of a data set since total change can vary between versions. I hypothesize that version analysis must factor in time when comparing change rates between versions. The change rate distribution of each AIM change follows a distribution significantly different from the version publication rate.

1.5 Hypothesis Statement

The work in this dissertation tests five hypotheses.

1. VersOn, compared to current provenance ontologies, increases the completeness of change graphs by capturing AIM changes.

2. VersOn reduces performance of change logs by no more than 50% where performance is measured by content per unit storage.

3. VersOn increases the precision and detail of change measurements compared to dot-decimal identifiers by enumerating changes.

4. VersOn, compared to the version publication rate, increases the accuracy of capturing the data set change rate.

1.6 Contributions

In Chapter 3, I present my first contribution, the model forming the basis of VersOn which is used to instantiate versioning graphs. Versioning graphs capture differences between objects, not the course used to create a data object, differentiating themselves from provenance graphs. The versioning graph enables my second contribution, a structured process to compute and publish linked-data change logs to test Hypothesis 1, covered in Chapter 4. The contribution eases consuming very lengthy logs, which data sets often produce, as well as enabling searchability and discoverability of changes affecting the version. My third contribution, discussed in Chapter 5, is using a versioning graph to provide a quantitative metric for determining change distance, a necessary component to answer Hypothesis 2. Differing methods between data producers and consumers in computing change distance highlights disagreements in the prodcuer/consumer dynamic which is demonstrated while evaluating Hypothesis 3. A fourth contribution is a comparison between version publication and change rates, found in Chapter 6, to determine the propriety of versions as proxies for change, used to test Hypothesis 4 and 5.

CHAPTER 2
LITERATURE REVIEW

2.1 Introduction

The data versioning landscape produces a variety of different approaches and standards towards change capture. Science agencies and organizations are only beginning to formally codify and standardize methods to capture and publish lineage information [17]. In comparing their methods, many systems also share the implementation of common versioning operations, suggesting an avenue for fundamental versioning properties. While Software Versioning Managers (SVM) prefer to adopt the dot-decimal identifier, Digital Object Identifiers (DOI) and other web identifiers contribute methods to connecting more expressive change documents. Change logs are a feature which commonly appears alongside software projects and provide insight in differences between versions, but they are found very rarely among data sets. Measuring the space between versions also appears under-explored in previous approaches.

2.2 Current Versioning Models

Version models provide a visual theoretical aid in understanding where a data object lies in relation to the rest of a work. The Atmospheric Radiation Measurement (ARM) group at Pacific Northwest National Laboratory used a model dividing the data into mathematical sets which versioning operations acted upon[18]. Adding files already in the set created a new set which inherited all non-intersecting files and included all the new ones. The model provided a means to organize and automate the versioning of ARM's daily expanding data sets.

The Health Care and Life Sciences (HCLS) Interest Group of the W3C recently released a model which resembles the FRBR model with a three level hierarchy of increasing granularity [19]. Their model, shown in Figure 2.1, separates the concept of a data set into three groupings. The highest level summarizes the data as an abstract work, perhaps better described as a topic or title. The data topic

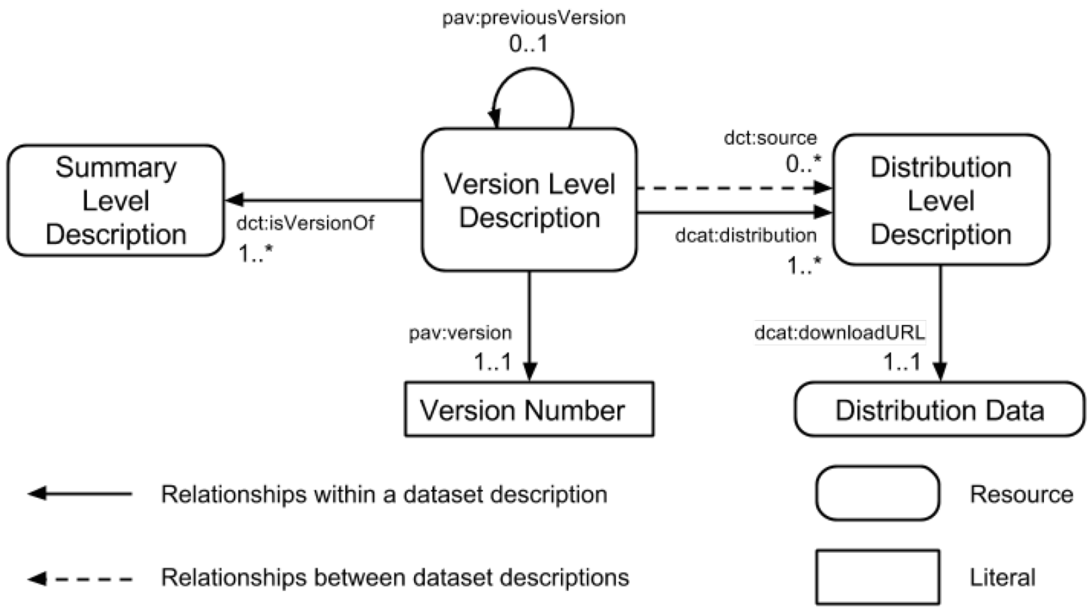

Figure 2.1: Data model from the W3C's Health Care and Life Sciences Interest Group separating data into three levels: works, versions, and instances. From Dummontier, et al. [19]

can have multiple versions over time. The version can then be instantiated into various distributions with different physical formats. The model—relating summary, version, and distribution—provides a method to implement the formation of FRBR's work, expression, and manifestation model on the Web.

From his definition of versions, Barkstrom also outlines a hierarchical version model as seen in Figure 2.2. The model features more intermediary levels than the HCLS's model by introducing data products and data sets as "homogeneous groupings" where differences can be introduced [14]. Each edge in the tree signifies a difference with other objects at the same depth, but the model does not provide a mechanism to explain the difference. The difference in the number of tiers employed in the HCLS and Barkstrom models also indicates that different applications will have varying expectations of granularity to their versioning models. Granularity is important because it defines the detail with which versioning activities will be executed. A general solution will likely need to be tiered and recursive in structure to accommodate different levels of specificity.

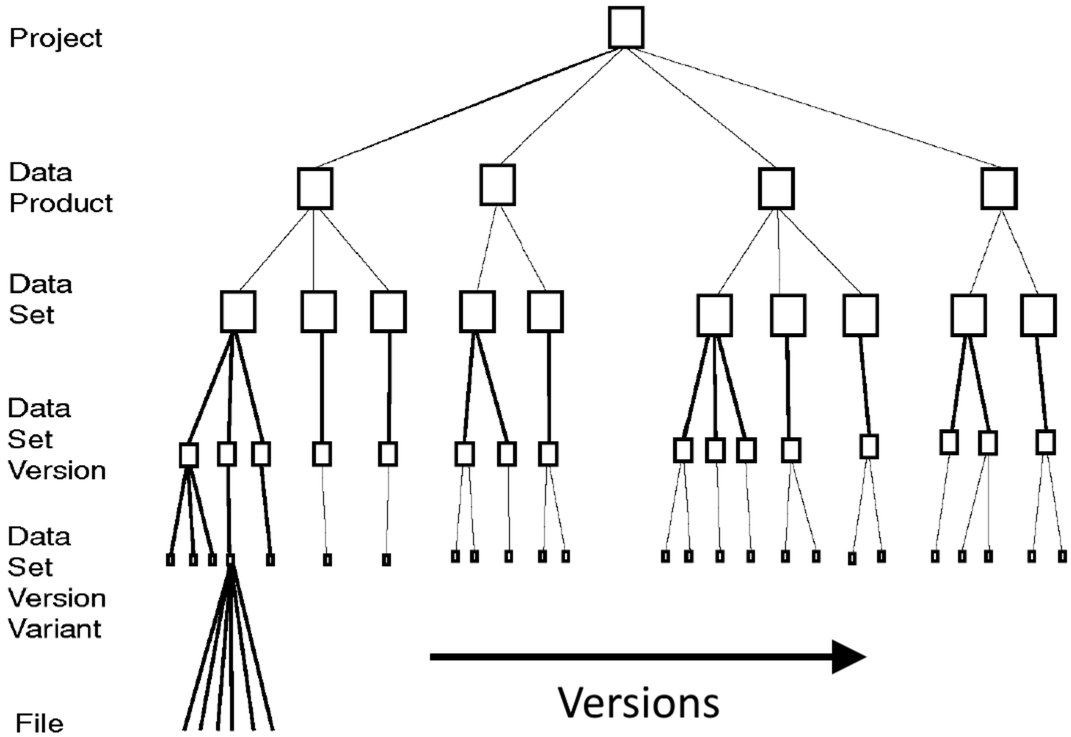

Figure 2.2: Visual representation of grouping hierarchy. Each row of the hierarchy defines a level of granularity where differences can separate "homogenous groupings" of data. From [3]

2.3 Documenting Versions

When workflow activities generate new versions of a data object, connective tissue becomes necessary to bind the versions. Change logs, artifacts resulting from the versioning process, play a major role filling in gaps between versions. Change logs document changes and explain, in human language, motivations behind the differences [12]. Introducing change log information into linked-data follows logically from the drive to improve transparency of characteristics connecting data sets, a linked-data principle. Very little evidence exists suggesting that change log data is being captured by linked-data practices. The provenance ontologies covered in Section 2.4 contain terms which are too broad to express changes at the same level of detail as change logs. NASA Earth Science's treatment of versions has been found to vary depending on a data community's traditions, meaning that if an ontology to express change data existed, the ontology may not be applicable to every community due to opposing traditions [14]. A review of data versioning in biology also identifies the lack of a standardized versioning format suggesting the lack of an accompanying standardized method to consume versioning data [5]. While some data sets will contain a change log, software projects have normalized their use in version release documentation [20]. The popular Software Versioning Manager (SVM) Git provides methods to attach messages to commits, but the messages are in plain-text rather than a linked-data compatible format [15]. As a result, software projects provide a basis for understanding the value of change logs to data sets with multiple versions. Drawing trends from change logs is limited by only appearing in human readable text. Wider adoption among data sets may be possible by making change texts machine readable.

The following studies demonstrate the value that can be unlocked by studying the behaviors and data stored within software change logs. Change logs play an important communication role in open-source projects, allowing new developers to trace implementation decisions made through the life-cycle of a project. In one project, researchers explicitly linked change log entries to documented bug reports, showing how bugs of different priorities were addressed [21]. The links give insight into motivations behind particular implementation decisions. Change logs linked with version releases also provide feedback to the user community that issues have been addressed, in addition to ensuring that improvements drive modifications to the code base. The rate of change log release has been shown to be a good indicator for the general health and activity within a software project [20]. The poor health of a project can negatively impact adjacent systems. A project mined change histories to determine dependency patterns in source code changes which would cause a cascade of follow-up bugs in other projects [22]. The research is pertinent to systems which rely on multiple projects to function, but also functions very similar to inter-operable data systems like those created by linked-data. The change history mining project diverges from previous approaches by looking primarily at the presence of differences between the source code instead of the natural language text in accompanying logs.

2.4 Provenance Representation

Linked-data is a collection of methods standards which improve data set interoperability by using technology to expose the characteristics and connections between collections of data on the Web of Data [1]. Provenance ontologies form a major section of linked-data approaches to data versioning. The coverage stems from the close relation between provenance and differentiating versions. The Proof Markup Language, one of the first semantic models to capture provenance information, expressed lineage relationships using inference reasoning through traceable graphs [23]. The technique provides a powerful way to express and imply sequences of relationships between different versions and characterize the manner of their relation.

2.4.1 Open Provenance Model

Open Provenance Model (OPM) was the product of a series of provenance challenges held at International Provenance and Annotation Workshops after participants began realizing and reaching consensus on the formulation and content of provenence information necessary to establish system interoperability [24]. In a project, the model has been applied to sensor networks, automating and unifying their provenance capture even as they grow [25]. To aid OPM's adoption, the framework Karma2 integrates provenance capture into scientific workflows and provides a more abstract view of their data collection activities [26]. The property *opm:WasDerivedFrom* constitutes a core concept in the model and marks the reliance of one object's existence on another object. For a large part, the relation encompasses the engagement which provenance models view versions, without further need to explore the derivation's content.

2.4.2 PROV-O

PROV-O, a W3C Recommendation, delineates a method to express data provenance in a more compact form as seen in Figure 2.3 [27] [28]. The recommendation uses a conceptual model relating activities, agents, and entities to describe data production lineage, the list of data ancestors leading to a data object [2] [29] [30].

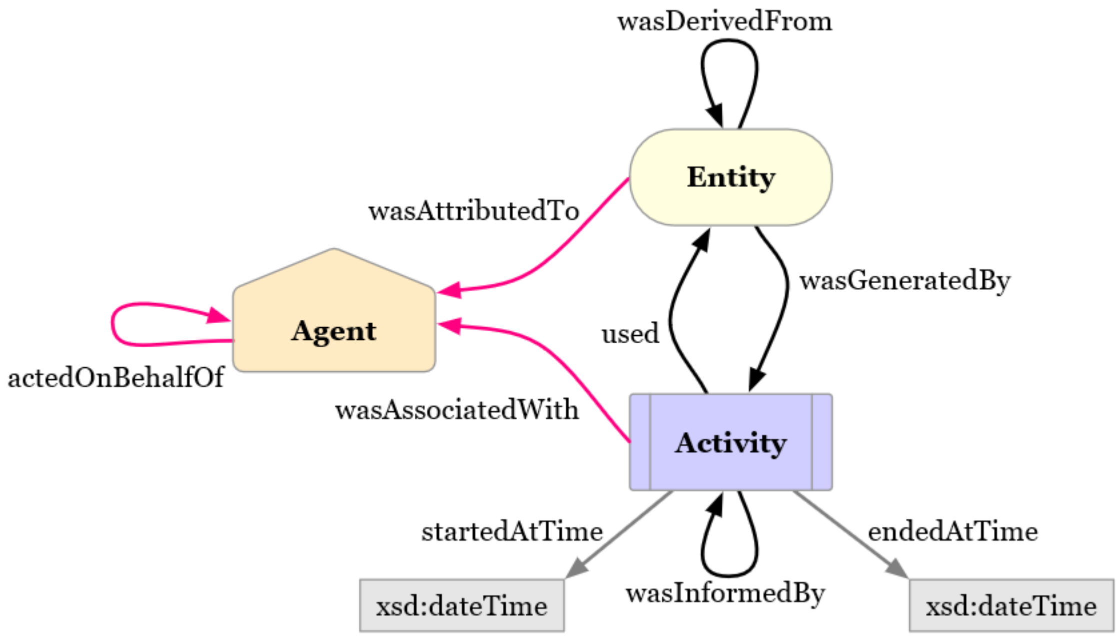

Figure 2.3: The PROV Ontology is divided into three main objects: Entity, Agent, and Activity. An Activity is an action or process which produces Entities which is associated with an Agent. Agents are usually individuals responsible for creating Entities. The high level ontology can capture complex provenance relationships to ensure traceability of data production. Figure 1 from [4]

Intended as a high-level abstraction, PROV-O takes an activity-oriented approach to provenance modeling. Every data entity results from the actions of some activity [31]. The conceptual model's expression occurs through the PROV-O Ontology (PROV-O), which is conveyed through various resource description languages [32] [33]. The ontology is further formalized into a functional notation for easier human consumption [34] [35]. One particular strength that has contributed to the adoption of PROV-O is its ability to link into other ontologies, making it easier for existing semantically enriched data sets to adopt PROV-O [36] [37].

PROV-O has provided a major contribution in maintaining the quality and traceability of data sets and reporting in the NCA published by the U.S. Global Change Research Program [11]. The inclusion into the NCA signifies that there is an increased likelihood of adoption through other scientific fields as a result of this reporting. The Global Change Information System, which houses the data used to generate the NCA and maintained by the U.S. Global Change Research Program, uses PROV-O to meticulously track the generation of its artifacts and results as they are used in the assessment report [38]. Usage means that not only does the data have a traceable lineage to verify quality, but the content of documents can have the same verifiability [39].

Komadu, a framework developed to alleviate workflow integration, utilizes PROV to improve upon its predecessor, Karma, by no longer utilizing global context identifiers that were not necessarily shared throughout the workflow. [40]. The framework and the adoption of PROV-O in high profile scientific applications means versioning applications can be overlayed onto a provenance skeleton once a versioning ontology has been created.

The PROV-O Ontology provides three different concepts that begin to encapsulate the provenance relationship between data versions. It defines a *prov:Generation* as “the completion of production of a new entity by an activity,” [4]. The definition means that the generation, which corresponds to adding an object to a version, must result from a *prov:Activity*. *Prov:Invalidation*, defined as the “start of the destruction, cessation, or expiry of an existing entity by an activity,” makes a similar connection between activities and entities [4]. A third concept, *prov:Derivation*, relates two entities, and the ontology defines it as, “a transformation of an entity into another, an update of an entity resulting in a new one, or the construction of a new entity based on a preexisting entity. ” [4]. PROV also has a property called *prov:isDerivedFrom* which conveys the same definition as a *prov:Derivation*. Using the property and concept together forms a qualified property which can be instantiated and further annotated.

2.4.3 Provenance, Authorship, and Versioning Ontology

The Provenance, Authorship, Versioning (PAV) Ontology is, “a lightweight vocabulary, for capturing “just enough” descriptions essential for web resources representing digitized knowledge” [41]. It provides a means to track versioning information through linked data by introducing *pav:version* to cite versions and *pav:previousVersion* to link them together in order [41]. The authors of PAV created the new property to expand on the Dublin Core concept *dc:isVersionOf* which is used when, "Changes in version imply substantive changes in content rather than differences in format" [42]. PAV supports the idea that a new concept becomes necessary to cover cases where new versions do not have to be substantive but can still be alternate editions of the original object. While it documents related versions well, PAV does not dive deeper in explaining the circumstances behind version differences.

2.4.4 Schema.org

The Schema.org structured data schema is not a provenance ontology but provides a means to supply searchable web pages with standardized micro-data. The schema has a collection of concepts which could be applied to versioning. The *schema:UpdateAction* is defined as, "the act of managing by changing/editing the state of the object," which encompasses the same responsibilities expected of versioning systems [43]. The terms *schema:AddAction*, *schema:DeleteAction*, and *schema:ReplaceAction* subclass the *schema:UpdateAction*. These classes model actions which further cement parallels between versioning and *schema:UpdateAction*.

Schema.org defines a *schema:ReplaceAction* as, "the act of editing a recipient by replacing an old object with a new object" [44]. The concept has two properties, *schema:replacee* and *schema:replacer* which indicates that a new object replaces an old one. Schema.org models the interaction by placing the replacement action at the relation's center. In comparison, the *schema:AddAction* is defined as, "the act of editing by adding an object to a collection" [45]. The action only involves the object and the new state of the collection, not involving any of the collection's prior lineage. Schema.org defines the *schema:DeleteAction* as, "the act of editing a recipient by removing one of its objects," [46]. The concept aligns well with other versioning systems, although deletion may be a strong assertion. The deletion would be inappropriate for use in cases where data is archived but no longer used, for example or when data becomes deprecated.

Provenance models are a necessary step towards the development of versioning models. Going back to the working definition of a version and the discussion from Barkstrom, versions are produced from changes to provenance, and without the ability to clearly trace provenance, versions cannot be clearly organized. A number of linked-data provenance models include versioning concepts such as OPM, PROV-O, and PAV. Each model approaches versioning differently, ranging from very broad to very narrow definitions of a version. None of the models approach capturing the differences between versions which is appropriate since the differences do not explain the processes used to create a digital object [24]. Versioning, in contrast, documents the results of using a different process.

2.5 Version Systems

Versioning systems take many different forms from Clotho, an application conducting versioning at the block level, to Champagne, a framework to propagate change data across multiple information systems [47] [48]. Each approach has a unique set of challenges to overcome. Closer to the data collection, version systems must be flexible and responsive to adapt to changing environments, but as the socio-technical distance of a repository increases away from the collection site, more formal methods are required to unify repositories [10]. Different approaches are also necessary to account for the needs of different domains. Versioning an XML text-file will need to account for serial file input and output as well as structured markup [49]. Many applications have adopted a tree-like structure which is further propagated by software versioning managers (SVM) [50]. The advantage comes from using well established graph theory methods, and applying the methods to an object's complex relationships in complex environments [51]. The growing population of web documents, however, presents a new smrgsbord of complicated data which will need scalable solutions [52].

2.5.1 Library Sciences

While many of the modern systems requiring versioning managers store digital products, libraries have been tackling similar issues for a much longer time. Libraries curate multiple editions of the same work, sometimes with significant revisions [8]. In many ways, versioned objects resemble multi-edition books or documents. Digital librarians have faced many challenges when searching for a persistent identifier due

Table 2 Suitable identifiers for each use case where solid green indicates high suitability, vertical yellow stripes indicates good to fair suitability; and orange diagonal stripes indicates low suitability

Identifier Type	**Unique Identifier**		**Unique Locator**		**Citable Locator**		**Scientifically Unique Identifier**	
	Dataset	Item	Dataset	Item	Dataset	Item	Dataset	Item
ARK	vertical yellow stripes	vertical yellow stripes	solid green	solid green	vertical yellow stripes	vertical yellow stripes	orange diagonal stripes	orange diagonal stripes
DOI	vertical yellow stripes	orange diagonal stripes	solid green	solid green	solid green	vertical yellow stripes	orange diagonal stripes	orange diagonal stripes
XRI	vertical yellow stripes	orange diagonal stripes	solid green	solid green	vertical yellow stripes	vertical yellow stripes	orange diagonal stripes	orange diagonal stripes
Handle	vertical yellow stripes	orange diagonal stripes	solid green	solid green	vertical yellow stripes	vertical yellow stripes	orange diagonal stripes	orange diagonal stripes
LSID	vertical yellow stripes	orange diagonal stripes	vertical yellow stripes	vertical yellow stripes	vertical yellow stripes	vertical yellow stripes	orange diagonal stripes	orange diagonal stripes
OID	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes
PURL	vertical yellow stripes	orange diagonal stripes	solid green	solid green	vertical yellow stripes	vertical yellow stripes	orange diagonal stripes	orange diagonal stripes
URL/URN/ URI	vertical yellow stripes	orange diagonal stripes	solid green	solid green	vertical yellow stripes	vertical yellow stripes	orange diagonal stripes	orange diagonal stripes
UUID	vertical yellow stripes	solid green	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes	orange diagonal stripes

Figure 2.4: Table of predominant identifiers used in science. From Duerr, et al. [54]

to evolving web technologies. Early citations referred to on-line documents using stagnant Uniform Resource Locator (URL), but this frequently lead to a condition known as link rot where moving the document would invalidate the URL [53]. Locators required a system to manage changes of old identifiers to new locations when people attempted to utilize references from print. The need eventually led to the development of Persistent Uniform Resource Locators (PURLs), which also suffered from link rot, and this eventually led to the distributed Digital Object Identifier (DOI) system used to track documents today [54]. The PURL used a centralized system that would translate dead links and redirect to a document's latest location. The system would still need to be manually updated, meaning links would rot if a document was lost or overlooked. DOIs rely on a network of managing agencies to collect and host submitted documents. In the specialized Handle system, the network has member agencies internally assign an unique name and concatenate it to the end of their host name. In Figure 2.4, DOIs represent the most suitable identifier used for citation in scholarly literature [54]. The DOI network provides a robust system to track documents, but when tracking data, it faces difficulty following the rate of change with more volatile data sets. Under current definitions, distribution organizations assign different DOIs to separate editions of a document. Documents often do not need new identifiers since they change very rarely as a result of the publication process. Data set production and distribution cycles move more quickly and react more sensitively to small content changes, including when data collection continues on after initial publication. Data set behavior becomes entirely too slow as data providers begin allowing users to dynamically generate data products from existing data according to their needs [55]. Some agencies have begun assigning versioned DOIs, but this has not become common practice. Other groups do not assign a new DOI, but reference the latest release of the document or object [56].

As digital methods have evolved, so have digital libraries. The documents that digital libraries store are no longer constrained by physical organization [57]. A book can physically be randomly stored for efficient retrieval, but the digital copy may reside in multiple locations depending on dynamic filters or search queries. The Mellon Fedora Project developed a standardized edition control structure to unify disparate digital library stores [58]. The regularizing edition tracking methods significantly improved the response time and relevancy of the library services.

2.5.2 Software Versioning

Software versions form the most visible displays of versioning often experienced by researchers. Version managers provide tools to archive and restore code through the development lifecycle. The Revision Control System (RCS), developed in originally in 1985, documents one of the earliest uses of the dot-decimal identifier [59]. This identifier uses a sequence of whole numbers concatenated by decimals. The system possessed many features of modern SVMs such as branches, a separate copy of the code for developing changes safely, which were identified by extending the dot-decimal identifier as seen in Figure 2.5. Not long after, the Concurrent Versions System (CVS) gained popularity with methods allowing multiple users to concurrently develop code to a central repository [60]. The most popular modern SVM is Git which also allows concurrent development but enables distributed repositories [15]. Each developer contributing to a project is considered by the system to possess the master copy of that project. The users collaborate by requesting and pulling other developer's master copies into their project. In previous SVMs, only the differences between software files were stored, but Git stores the entirety of each

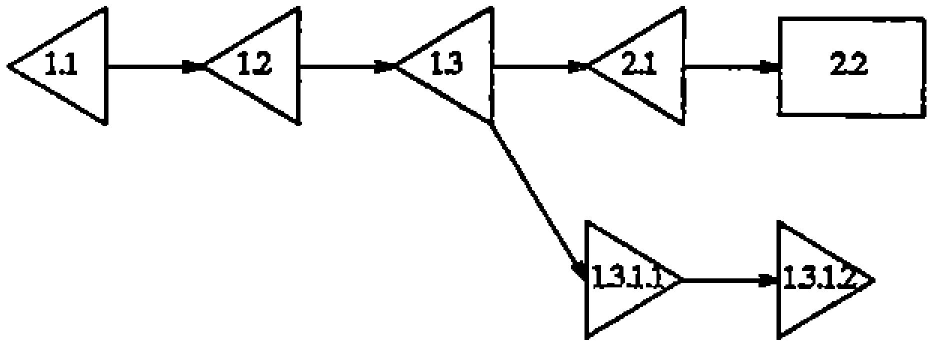

Figure 2.5: Commit history of an object in RCS with changes in the main line stored as back deltas and side branches stored as forward deltas. Figure 5 in [59]

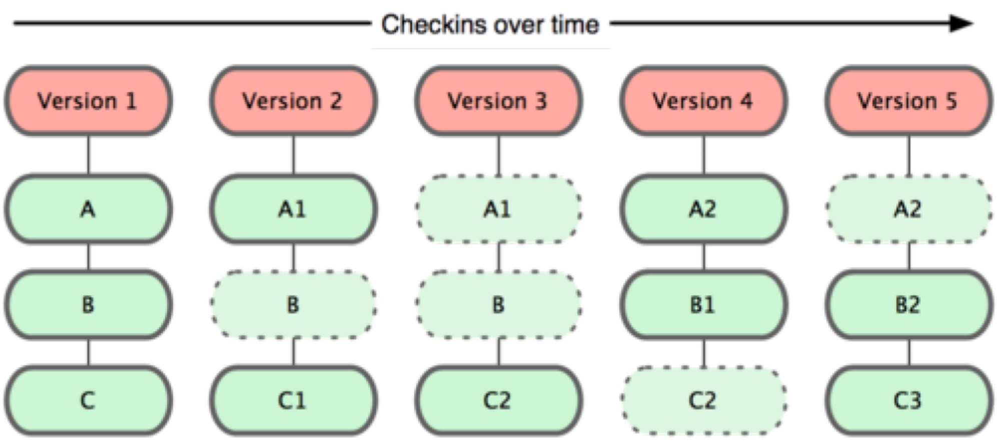

Figure 2.6: Git stores changes in the repository as snapshots of individual files. Figure 1.5 from [15]

file version. Figure 2.6 demonstrates an example of how Git employs storage space for multiple versions [15]. Only a pointer is stored in subsequent versions for unchanged files, saving space. Fischer, et al., demonstrate the importance of software version systems by integrating the manager with a bug tracking system to indicate the bugs a version release addresses [61].

2.5.3 Database Versioning

The need for data versioning methods grew alongside the growing popularity and power of relational databases. Klahold, et al., introduced using abstract versioning environments in 1986 to separate the temporal features and organize the data into related groupings [62]. Research in the versioning area focused primarily on the database schema. The results were temporal databases where schemas included time and dated transactions modifying the schema [63]. Temporal databases allowed old queries to be executed on updated schemas, improving the reproducibility of results. Capturing periodic snapshots or copies becomes unfeasible with increasingly large centralized database systems. Data collection continues to migrate towards massive data warehouses which store and serve a wide variety of data [64]. Proell and Rauber have investigated tracking data queries instead of the database as a more scalable solution to reproduce data [65]. The queries can then be used as publication citations to provide scalable, reproducible references to older data [66] [67].

2.5.4 Grid Versioning

The grid provides a sensitive environment for versioning where there are many users and data movement across the grid should be avoided. The CERN grid for the Compact Muon Solenoid experiment carefully developed processes which allow references by multiple users to the same file without copying that file across the grid [68]. Versions lock and release to permit parallel processing while still archiving additions and modifications to the data. Grid versioning applications also begins to highlight the difference in versioning usage patterns between users and producers [16]. Deeper exploration into the A Toroidal LHC ApparatuS (ATLAS) system documentation did not reveal specific use cases explaining the differences. The grid also provides users with the ability to begin dynamically defining data sets to their needs by aggregating results from across the network [55]. The process would create new data sets without prior existing change documentation and fueled a demand for responsive frameworks which could track the discordant data collection conditions assimilated by the system [69].

2.5.5 Ontology Versioning

Ontologies play a major role in defining domains, especially in the biological and medical fields where terms and definitions can change rapidly across highly variable organisms [70]. As a result, the ontologies require consistent methods to capture and model changes to evolving terms. Tools aid in the process by detecting differences between ontologies [71]. Klein and Fensel have found that when the changes are discovered, both forward and backward compatibility must be established for clear ontology versioning [72]. Not only must the path from an old term to a new one be clear, but a method for new terms to interact with old data must also exist. They additionally identified three levels at which ontologies can differ: the domain, the conceptualization, and the specification. Hauptmann et al., define a method to version ontologies natively within a triple store using linked data [73] [74]. The method heavily relies on the context of stored data.

2.5.6 Evaluation

Versioning systems cover a wide variety of different application environments, and each uses terminology to define versions in the context of their particular domain. Application based systems such as software and grid versioning focus primarily on identifying large, medium, and small differences between versions. The size approach suffers many drawbacks as a result of variety in versioning environments. Small changes logged in Clotho would barely register in massive systems operating on the grid. The requirements to differentiate changes is not universal across versioning systems. Other than software version managers, the systems do not incorporate methods to include change logs. They use the existence of an alternate version as sufficient explanation for what has changed. Bose, Frew, and Tagger all recognize the need in versioning for a standardized representation, but each domain defines change according to the needs of their application [75] [5]. In isolation, the systems to not recognize the commonality in utilizing similar operations to conduct versioning activities.

2.6 Data Versioning Operations

Among all the systems surveyed in Section 2.5, every one employed some form of the operations add, delete, and modify. Literature surveys often expect versioning systems to interact with data uniformly because they are asked to perform the same functions [5]. Different data sets, however, may utilize each of the three core operations at different rates [76]. The differences help to characterize the data set in ways such as a growing set with many additions, a stable collection featuring occasional corrections, or a wildly volatile data set consisting of often deleted and replaced data files. Understanding these would give insight into the maturity and health of a data set.

While data addition and modification remain fairly uncontroversial, there is a mild division between practical and theoretical approaches to data deletion [47]. A removed object provides evidence of an erroneous activity's results or intermediary steps leading to a final product. As a result, version management should maintain and track invalidated data instead of deleting it. The software versioning manager Git uses a method of compressing older data to conserve space without deleting the data [15]. Available storage space places pragmatic constraints on the number of projects which can adopt snapshotting practices. In applications which cannot recover erroneous data nor use it as documentation artifacts, like corrupted surveillance images. Some high energy physics experiments cannot re-collect observational data due to cost, and as a result, they cannot replace or re-process poor quality data [9]. While the distinction between 'deletion' and 'invalidations' remains largely semantic, the terms' use in this document reflects an understanding of the different constraints and requirements placed on versioning systems. As a result, invalidation is adopted as a broad, general term to also encompass data deletions.

A handful of other operations exist among version managers, but they do not prove ubiquitous across most applications. Software versioning tools like RCS commonly feature branching and merging functions to create a versioning line separate from the stable master branch [59]. Branching mostly provides an organizational role in development by allowing developers to experiment without contaminating a stable software release. Figure 2.7 models a branching operation, showing versions C3 and C5 in branch iss53 before being merged back into the production line as C6. Branching allows for more orderly management of versions, but does not conduct versioning itself. Other activities provide functional operations such as locking and unlocking files from edits to prevent race conditions in branch mergers. Locks does not introduce any new relationships but allows the tool to operate more smoothly. Many version control tools, likewise, include functions to display the versioning tree, but this is also an ease-of-use function [51].

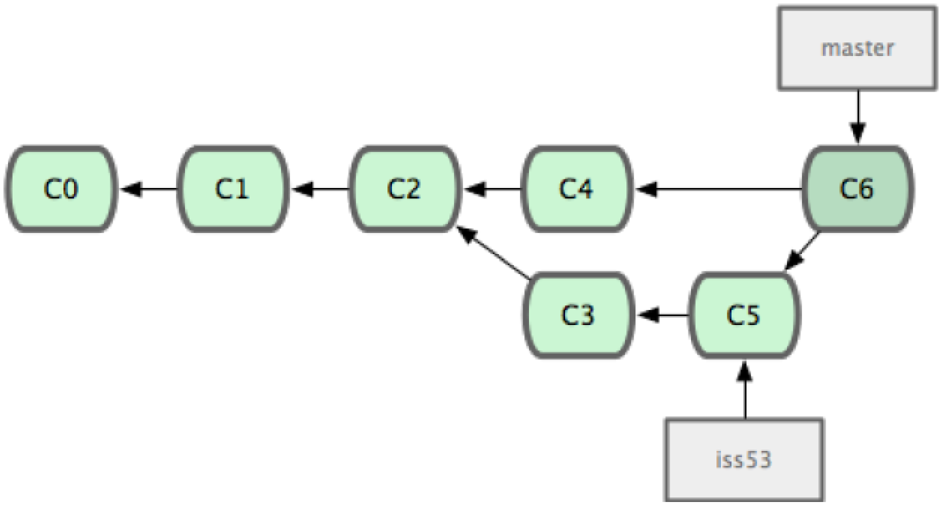

Figure 2.7: Example of a commit history with branching stored in Git. Figure 3.17 from [15]

2.6.1 Types of Change

Another commonality across many versioning systems is differentiating between major, minor, and revision changes. Definitions for what constitutes each category differs across applications, but the desire to do so often stems from the tradition of 3-number dot-decimal identifiers. Barkstrom uses the ability to scientifically distinguish between two data sets as a criteria for major divisions among groupings [3]. At lower levels, he notes that science teams can no longer discern scientific differences between data sets. They observe that, instead, changes to format and structure contribute significant alterations without changing any values withing the data. As a result, these technical changes form a second boundary to meaningfully separate minor version groupings. Finally, the explicit values may need occasional revisions to correct lexical errors such as spelling or formatting. Data producers will often use qualitative measures to determine the type of change occurring between versions. Versioning system users wish to achieve insight into the type of change that occurs between versions.

The exact category that a particular change falls into can be controversial. The decision to provide concentration units from parts per million to milligrams per milliliter poses a Technical change for a data producer. However, for a data consumer, the alteration may be viewed as a Scientific change as it invalidates the methods they had previously used. The conflict in view illustrates the data consumer-producer dynamic. In general, data producers control the versioning methods, but data consumers determine a change's impact through use. Producers tend to use versioning systems to ensure data quality of service through audits and recovery tools [9]. Meanwhile, a consumer will analyze the historical changes and determine the impact this may have on their data use. As a result, this means that data versioning systems must communicate a dynamic view of the changes in a system contextualized by the user of that data.

Version managers often disagree at the point many technical changes sufficiently modifies a data set that it comprises a scientific change. As determining changes in science requires expert understanding over a domain, different measures should be explored to address the distinction.

2.7 Identifiers

The most widely identifier scheme associated with versioning is the dot-decimal identifier [50]. Whenever, a new version is made, it receives an identifier with one of the numbers incremented as seen in Figure 2.5. Such a procedure fails to communicate the extent of a change because, regardless of the amount, the identifier will increment only one number. Changes to the left-most number often signify a more important change. Many software applications use the 3-number Major.minor.revision format in labeling software releases. Numbering the version this way, however, does allow computers and readers to quickly parse the version name and discern that a change has occurred, but not much value exists beyond that [51]. Most importantly, it groups together changes from the lower spectrum of minor or major change with those in the upper, more impactful, changes. Obtaining a clear characterization of a version change is difficult without a longer series of numbers. In addition, version numbers capture the overall change of a data set, but users may not interact with collections that way, only caring about parts of the data or certain kinds of change. There is also little standardization or formal requirements in naming methods. Ubuntu utilizes a dot-decimal version labeling scheme where the two number identifier corresponds to the year-month values of the release [77]. A common method used to address the distinction between versions is a human-readable change log, further discussed in Section 2.3.

The discourse on DOIs highlights the importance of understanding the limitations of particular identifier schemes. With respect to Figure 2.4, no identification scheme fits the description of a scientific identifier. Duerr, et al., define a use case to make the argument that scientifically unique identifiers are necessary, “to be able to tell that two data instances contain the same information even if the formats are different” [54]. A possibility to consider is that identifiers may require incorporation into a data model to discern between scientific differences. An identifier works well in revealing the characteristics of an individual object, but it should not be expected to explain its relationship with other objects. A data model provides better insight into the different roles objects play in a relationship. DOIs also provide a new means to identify versions using Uniform Resource Identifiers (URIs) which can be dereferenced to provide change information or the data depending on the context.

Using identifiers to convey extended versioning information becomes more difficult with the adoption of distributed version managers like Git [60]. Each participant in the federated repository is the master of their personal copy of the code. Upon completion of their distribution’s part, they may request that it be pulled into another participant’s distribution. While each developer’s individual repository can follow a linear identifier scheme, the identifiers would not work as the overall project bounces around different primary repositories with mismatching sequential identifiers. The dot-decimal identifier scheme could be made to work in such an environment by severely limiting the distributed manager’s utilized features. Figure 2.8 illustrates a workflow which utilizes distributed repositories to manage very active public software projects. Each lieutenant developer manages a section of the overall code, and they dampen the number of requests made to the dictator by collecting changes and submitting them over longer intervals. As a result, relying on identifiers to convey and contain versioning information limits the evolution of new and valuable methods of processing change in digital objects.

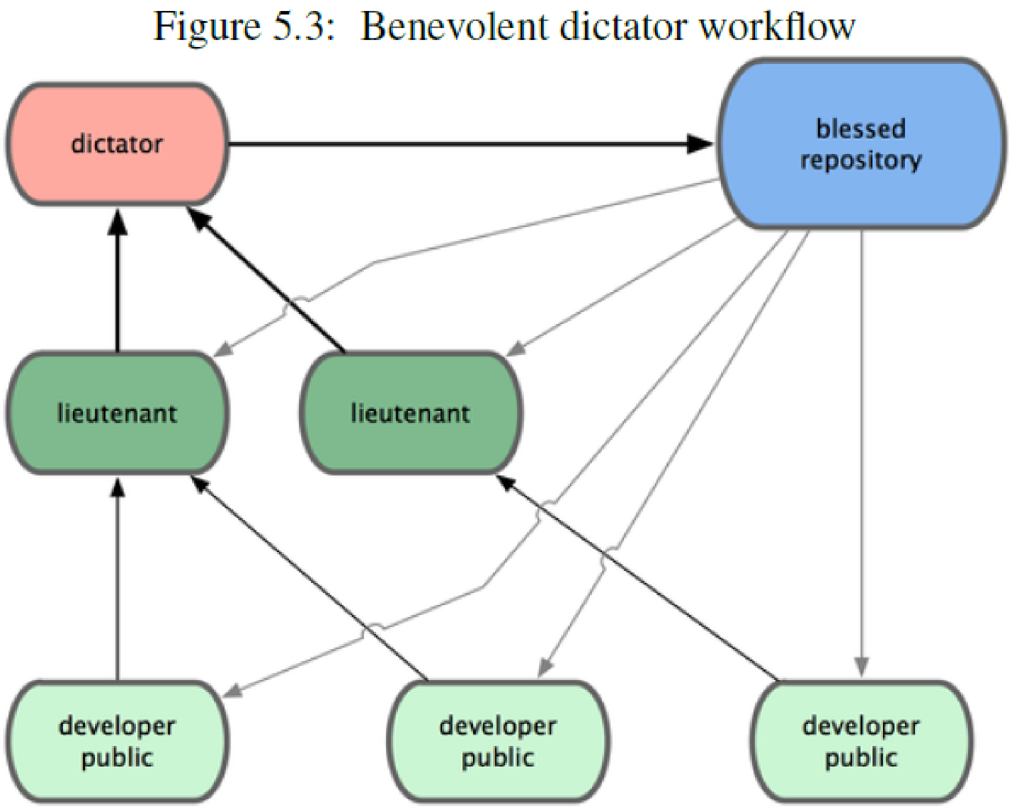

Figure 2.8: A distributed workflow to control for volatile versioning behavior. From [60].

2.8 Structured Data

The Resource Description Framework in Attributes (RDFa) framework encodes linked-data vocabularies into HyperText Markup Language (HTML) documents, and provides an opportunity to make change logs machine interpretable. [78]. Figure 2.9 illustrates the semantic difference between what web crawlers and what humans see when they consume web pages. People intuitively understand that certain strings represent meaningful information based on location and style. RDFa seeks to encode that understanding natively for effective machine consumption. Extending this approach into publishing change logs, will allow linked data to capture the metaphorical meat of change content.

The implementation requires changing publishing practices from plain-text documents to something structured-data compatible such as HTML. The change also has the added benefit of making the logs available on-line, and thus, more

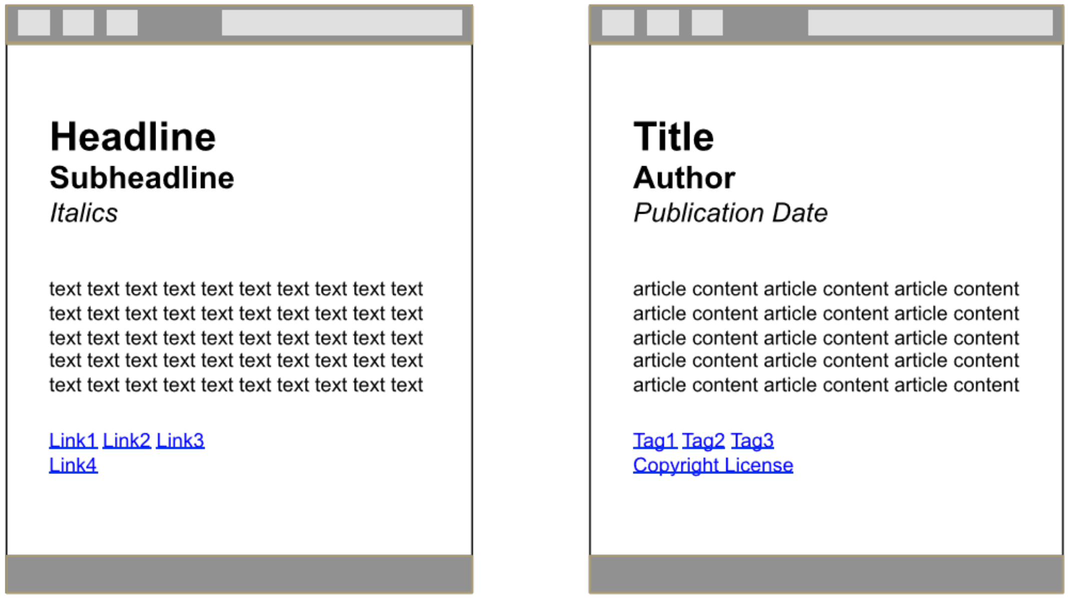

Figure 2.9: Illustration of the difference in what autonomous systems see when crawling a web page and what humans see when reading the same material. Figure 1 from [79]

openly accessible to data users through the utilization of web based search engines. Large companies such as Google have already begun equipping their web crawlers to consume structured data such as RDFa from web pages. RDFa has already had significant success in adoption across a variety of web publication platforms and eases the search for their content [80]. The design of RDFa focuses on describing the web page's content through markup [79]. The underlying or resulting versioning data model may not conform with the format of content presented in the change log. Poor affinity would lead to a poorly structured graph or missing content, undermining the value gained by encoding linked data into the change log. As a result, another method using JSON for Linked Data (JSON-LD) was pursued since its purpose is to store data separate from visible content.

The JavaScript Object Notation (JSON) data format allows web pages to store data for JavaScript applications within the document. It utilizes a simple and robust syntax to accommodate a wide variety of content. JSON-LD extends the original specification by defining rules which allow entries to resolve as web vocabularies, giving them a meaningful context [81]. Because it stores data separate from visible content, JSON-LD does not need to adhere with the constraints of visible content. Every linked data triple must instead be explicitly defined, meaning that resulting documents may likely be much larger than their RDFa counterparts.

2.9 Change Distance

A major function of versions is to communicate the amount of change which exists between two versions. The quantity plays a major role in determining the freshness of data within a collection, indicating its pertinence to new projects [82]. Additionally, changing versions are often used to signal other applications downstream that a new version may be necessary to adopt data improvements [83]. Many efforts currently to compute a distance measure relies on data provenance. Formalizing operations on provenance remains an active field of research [84]. Other approaches relate to determining semantic similarity in trying to summarize the data set and computing a distance measure [85].

2.9.1 Provenance Distance

Previous endeavors to extract insight into data set performance or behavior using provenance have provided exciting results [86]. The research, however, generally studies the current state of an object's provenance rather than compare two provenance graphs. As stated previously, versions result from slight variations between the provenance of two objects. The connection suggests that studying the variations' magnitudes will help predict the change's impact. The measurement known as provenance distance seeks to determine the impact of changes in provenance on new data versions through measuring graph edit distances.

The first ingredient necessary to calculate provenance distance is a linked data graph capturing the sequence of events leading to the old and new objects' creation, like the one shown in Figure 2.10. The graph shows the multiple lower level products involved in creating a Level 3 ozone indicator. This can be accomplished

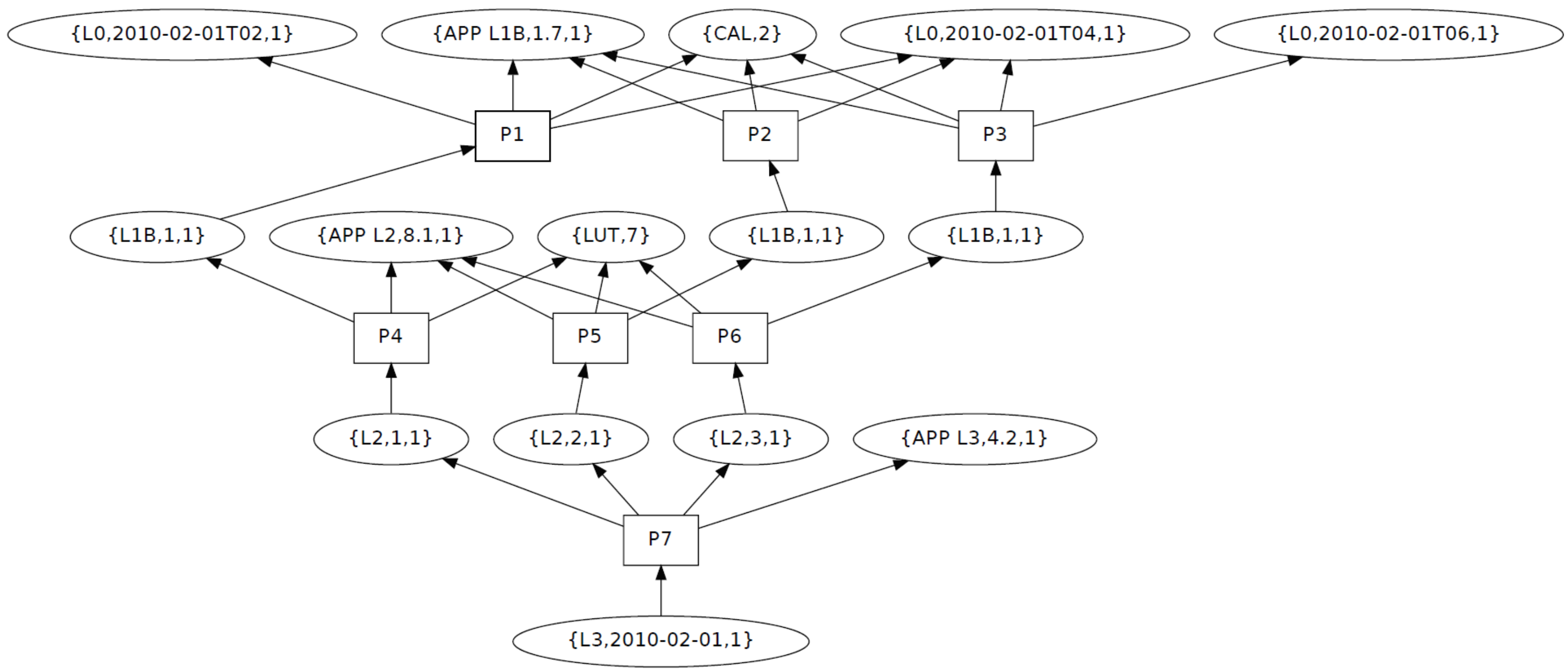

Figure 2.10: Provenance graph of a Level 3 data product, showing the inter-relations between different data products in generating the final product. Figure 2 from [83]

through the use of previously mentioned provenance models, but these graphs are not widely available. Using PROV to represent provenance data in a semantic model produces an acyclic directed graph with labeled nodes. As a result, the provenance distance problem reduces to similarity measurement. When calculating the similarity measurement of two graphs, algorithms determine how far the graphs are from being isomorphic [87]. Node labeling simplifies the similarity measurement process by providing nodes which must match together, and greatly reduces the complexity from computing generalized graphs. Graph Edit Distance, counting the edits necessary to transform one graph into another, provides a quantitative measure to associate with this process [88]. Some variations count edge changes [89].

In Figure 2.11, the left graph transforms through a move of edge 1 and a rotation of edge 4, resulting in an edit distance of two. Such changes in a provenance graph would demonstrate an alteration in dependencies between objects used to generate a final notable product. Isolating changes responsible for differences in provenance can become difficult in complex environments as Tilmes observes in 2011,

> Consider the relatively common case of the calibration table, which is an input to the L1B process, changing. Even though the version of the L2 or L3 software hasn't changed, the data files in the whole process have been affected by the change in the calibration.

[83]. L-number is shorthand for the level system featured in Figure 1.1. While provenance distance may be straight-forward to calculate, the indicator hides many insights into an object's behavior.

Methods to provide quality of service boundaries leveraging provenance already exist which compare workflows based on performance criteria [90]. These procedures focus primarily on quick retrieval and efficient storage instead of capitalizing on the latent information accessed by reasoning across data set versions [91]. Using only provenance data is insufficient to give insight into a change's impact because it does not provide information on structural or content differences which is what change logs provide. Measuring a change's impact with accuracy comparable to a change log requires a more detailed understanding and description than

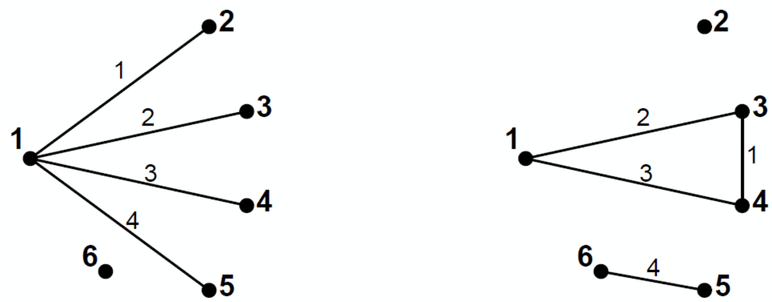

Figure 2.11: The labeled graph on the left transforms into the right graph under two edge edits. Figure 2 from [89]

provenance can provide [75]. Sufficiently precise versioning measurements cannot be provided by provenance distance, but it could indicate the confidence of versioning results, which is out of scope for this project.

2.10 Summary

In order to better formalize data versioning information, an approach must be developed leveraging common aspects of very disparate versioning systems. A data model based around versioning operations instead of impact remains largely untouched across the field. Version identifiers must additionally be untangled from communicating change distance which change logs accomplish with greater detail. The logs, in turn, need to be extended for machines to consume, easing adoption as data set size grows through automation. Change measures utilizing version graphs rather than provenance graphs are also under-explored. Chapter 3 presents a model to create a versioning graph.

CHAPTER 3
VERSION MODEL SPECIFICATION

3.1 Introduction

A versioning data model needs to address a variety of needs not met by provenance models as indicated in Section 1.1. In PROV-O and PAV, the modeled entities are exclusively one-dimensional with each version leading sequentially to the next one. The HCLS model, Figure 2.1, and Barkstrom model, Figure 2.2, however, display a more complex two-dimensional hierarchy. The tree models better capture the tiered granularity separating different versions which can result from a higher-tier macro change. These models also tightly couple new objects with changes to their underlying attributes. The tiered approach more clearly explains the scale on which two objects within the tree differ.

Provenance models provide concepts to sequentially order data objects but lack the ability to convey differences between farther spanning objects. In Figure 2.2, the left-most leaf node and the right-most leaf node differ by three changes at the data product level. A provenance model would need to rely on qualified properties to connect further annotations and describe the higher level changes. Remember that a common function of versioning systems is to provide a method to determine the amount of change or difference between two objects of a work. Much of the differences become lost when compressed into a single relation in a provenance graph. Additional annotations are often in natural language and do not provide a regular attribute to quantify.

The provenance models, on the other hand, do a much better job in explicitly defining the connection between objects which the tree models imply with structure. The versioning model must contain a mechanism to convey how changes to parts of an object contribute to that object's transition into a new version. The fundamental operations—addition, invalidation, and modification—are used by the model to capture change in a more detailed manner. These details provide a mechanism to measure change between versions with better clarity than current methods.

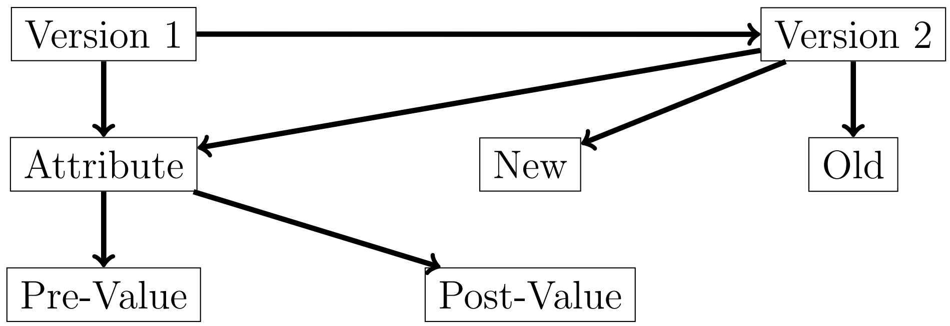

Figure 3.1: Provenance based versioning model.

When making change logs more approachable for usage in data sets, two options are available. The first approach continues writing change logs in only human-readable language and relying on advances in natural language processing to allow computers to read the change logs. The second approach uses linked data to encode the change log with machine-computable statements. Since natural language processing is currently not sufficiently articulate, the second approach is taken. In doing so, a versioning data model needs to be developed which can capture changes in the way a change log organizes information.

3.2 Exploring Possible Approaches

The first approach, which will be referred to as the provenance based model and seen in Figure 3.1, simply extends the provenance relation with additional concepts to capture more types of relationships. Until the introduction of or comparison with Version 2, none of the concepts in Version 1 can be considered new or old. As the responsible party for introducing changes, Version 2 becomes associated with *New*, *Old*, and modified attributes. Version 1 also has a link to altered attributes since it provides the *pre-value* used to contextualize Version 2's *post-value*. The pre- and post- values are included so that a user can see how much the attribute has changed, much like with a change log.

Adding the attributes as concepts to the provenance based model addresses PROV's and PAV's flat approach to version relations, but the attributes do not capture the inter-relation between objects for *New* and *Old* attributes. Having Version 2 be responsible for all the changes causes issues with the model since it must

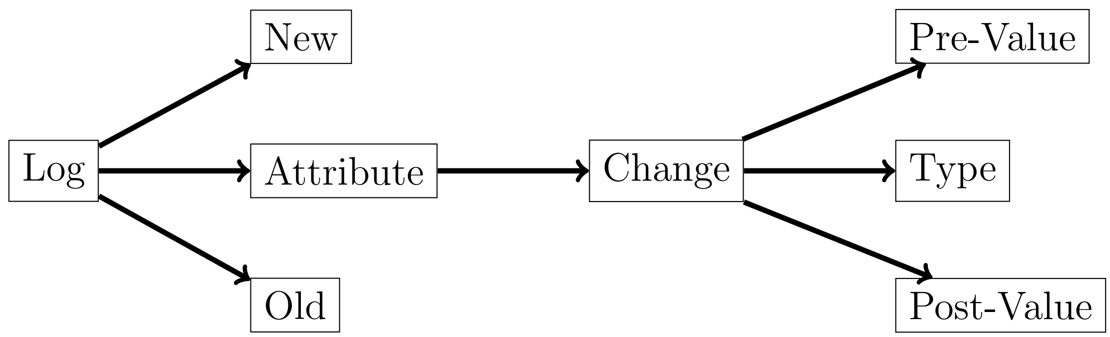

Figure 3.2: Change log based versioning model.

be associated with attributes from an entirely different object. The *Old* attributes do not appear within Version 2, failing to meet the version-attribute connectivity requirement. Associating *Old* attributes with Version 1 would be more appropriate and intuitive to understand. To test whether the provenance based model satisfied version continuity, a hypothetical third version is added. For attributes which are modified in Versions 1, 2, and 3, all pre- and post-values are grouped under the singlar Attribute, making the sequence of values ambiguous. Because the provenance based model cannot unambiguously accommodate a third version, version continuity is not upheld.

Using change logs as a basis, a log based model, shown in Figure 3.2 was created. Attributes are attached to the log as the primary indicators for old, new, and modified concepts. Change logs often show a break down and group changes by attributes. For modified attributes, an additional change concept is associated, encapsulating the values and nature of the change. At the far right side of the figure is a concept called *Type* which indicates more specifically the nature of the change, for example one unit of speed to another. Pre- and post- values were also included to explicitly define the change concept.

The primary drawback of the log based model is that the Log concept is the origin for every modification to the data even though the document only reports the differences. The version objects are also left out of the log based model, leaving the Log concept in possession of the attributes. One of the major breakthroughs with the log based model formulation is that while specific values are kept in the log, pre- and post- values do not need to be in a versioning model. By encoding

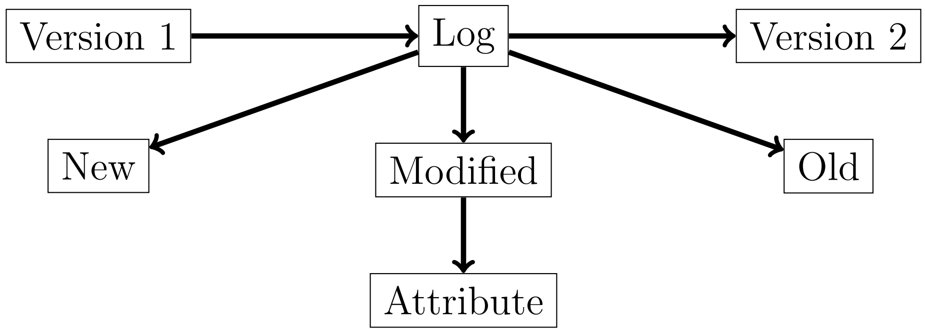

Figure 3.3: Hybrid provenance and change log versioning model.

the type of change, the need for actual values becomes superfluous as change type is more generalizable across domains and contexts.

Figure 3.3 combines the provenance and change log approaches, resulting in the hybrid model, by modeling the change log as a transition between Version 1 and Version 2. The idea enables distance capture between versions by encapsulating all changes within the Log concept. The changes are then associated with specific attributes. Pre- and post- values do not appear in the model because, as explained previously, the presence and kind of change is more valuable information than assessing the explicit values involved. As the number of changing entries increases, more values would need to be stored, resulting in the copying of the data rather than summarization of the changes. Notice in Figure 3.3 that *Attribute* has now become disconnected from either Version concept. Reconnecting the *Attribute* concept brings into question which version it should be associated with since it exists in both. The larger issue with both the log based model and hybrid model is that the versioning model resembles a tree more than a graph, making linked-data queries less powerful, as most of the concepts are disconnected.

A fourth formulation, in Figure 3.4, leverages the insight that when a **change** interacts with an attribute, the attribute has changed in the next version, resulting in the model referred to as the fully connected model. The fully connected model addresses the attribution problem by forming two attributes, each associated with a different version. These attributes inform a **change** which acts upon both *Version* concepts. The *Log* object is dropped for the fully connected model since it is a method to convey change and not an actor involved in the change. The highly connected model is not able to capture new and old *Attributes* because Attribute 1 and Attribute 2 cannot be removed from the model, meaning that the fully connected model cannot represent all additions, invalidations, and modifications.

One observation is that the relation from changes to versions is redundant since the links from *Version* to *Attribute* to *Change* implies the same relationship. Removing the explicit relation would shorten the number of triples required to encode a change and improve scalability. The versioning graph using the highly connected model would also be easier to query if the edges were oriented in the same direction, additionally implying that change flows from one version to the next. These final observations result in the final versioning model which is used to instantiate VersOn.

3.3 The Final Model

The final versioning model incorporates three kinds of objects: versions, attributes, and change. A version object represents the items being compared such as a book or spreadsheet. In PROV-O, a version would likely correspond with the *prov:Entity* involved in a *prov:wasRevisionOf* property. The **attribute** object refers to specific parts which make up a version. Attributes could be lines in a book or columns in a spreadsheet. Including attributes addresses the lack of detail involved in a *prov:wasRevisionOf* or *pav:previousVersion*. The relationship between **versions** and **attributes** captures the influence that changes in the underlying part will have on the overarching version. Because the model refers to specific parts of a version, the version concept corresponds most closely with a FRBR **manifestation** rather than an **expression**. The presence or absence of an attribute is used to determine the kind of change which occurs to the attribute between versions. Change are used

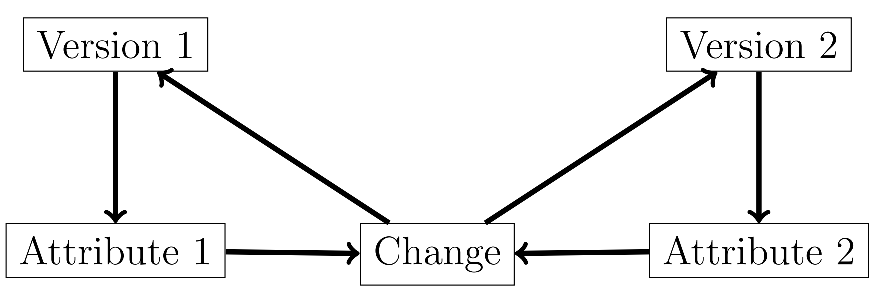

Figure 3.4: Highly connected model of just versions, changes, and attributes

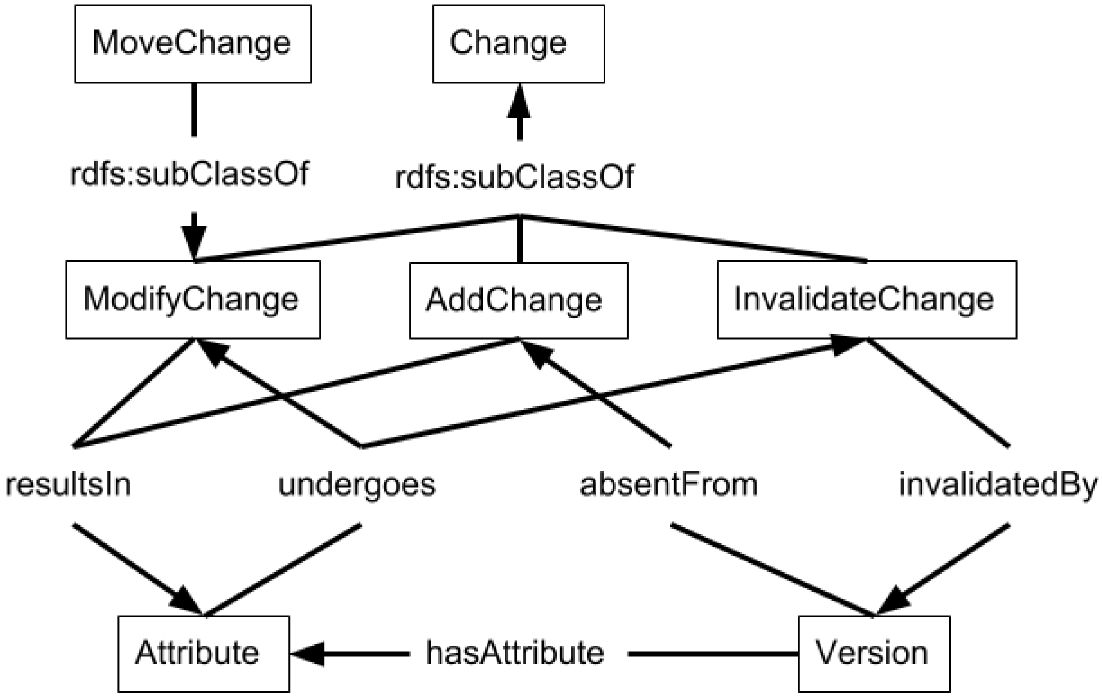

Figure 3.5: Ontology map of the Versioning Ontology. The primary concepts are Version, Attribute, and Change. Changes are subclassed into disjoint AddChange, InvalidateChange, and ModifyChange concepts. The MoveChange is a subclass of ModifyChange, demonstrating the extension of the core change concepts.

to link together attributes from different versions. The change captures a difference between the old version state and the new version state. While the change object greatly resembles a PROV-O qualified property, its form can change depending on the kind of change, like a *schema:UpdateAction*. The final model is implemented by The Versioning Ontology (VersOn). The full ontology map can be seen in Figure 3.5.

3.3.1 Left-hand Right-hand Convention

In the following diagrams and figures, the original or base version and its attributes will be placed on the left-hand side and the new version will be placed on the right-hand side with its attributes. References to the versions as previous and next are avoided since sequencing may not play a major role in distinguishing versions. Scientific data in large repositories often track sequential releases of data, but

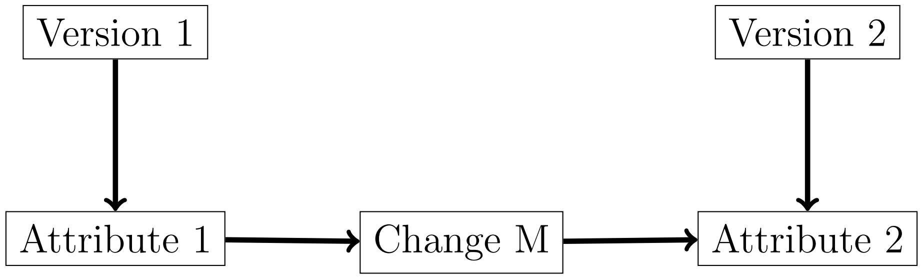

Figure 3.6: Model of the relationships between Versions 1 and 2 when modifying Attribute 1 from Version 1 as a result of Change M, resulting in Attribute 2 from Version 2

a book may have different versions distinguished by printed language. To recognize the distinction, objects will be referred to as the **left-hand version** or **left-hand attribute** when they are not sequentially or temporally related.

3.4 How Changes are Represented in the Model

The final model bases changes around the three core versioning operations because their commonality across systems provides a fundamental basis for comparisons. additions occur when an attribute appears only in the **right-hand version**. When an attribute only shows up in the **left-hand version**, the final model captures the relationship as an invalidation. Finally, a modification change has attributes in both the **left-** and **right-hand versions**, but the modification change only connects two attributes if the values are different. These three combinations cover the possible situations within the final model.

3.4.1 Modification

The modification relation occurs when an attribute appears in both versions and their values are different. In Figure 3.6, a modification is captured between two versions. Each version has an attribute, Attribute 1 and Attribute 2, respectively. Finally, a change object connects the two attributes, denoting that the values described by the attribute are different.

The specific values pertaining to Attribute 1 and Attribute 2 are not captured by the model because acknowledging that a difference exists is more important. Extending the model to properly communicate the significance of a modification

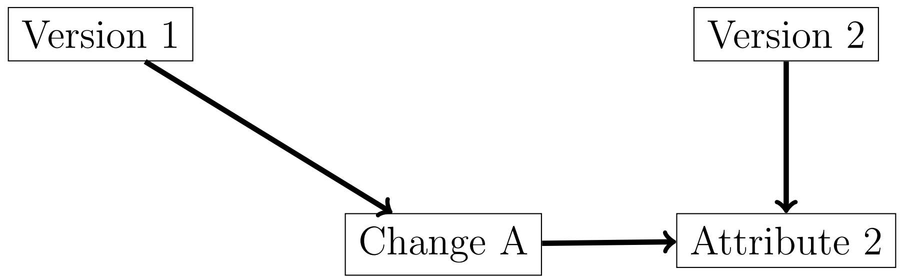

Figure 3.7: Model of the relationships between Versions 1 and 2 when adding an Attribute 2 to Version 2 as a result of Change A

for a wide variety of domains would require sizable domain knowledge and would be outside of a data versioning scientist's domain. In addition, the model would essentially begin storing a copy of the data set, leading to space and redundancy concerns.

In some applications, a **modification** is represented as an **invalidation** followed by an **addition**. The representation has a couple of problems. The first is that the sequence of changes implies that there is an intermediary stage where all the modified values have been invalidated but not added. The intermediary stage constitutes a new version of the data object, even though only one change is being made. The second issue with using the two change sequence is that the two states of attribute, before and after the **modification**, becomes disassociated. Any new attribute being added to the version is not necessarily associated with an attribute in the left-handed version. Establishing the association between attributes after an **invalidation-addition** sequence is the same as and more concisely expressed as a **modification**.

3.4.2 Addition

In Figure 3.7, the addition model differs from the modification construction by the absence of Attribute 1. The absence creates a disconnect between "Version 1" and "Change A". We know that a connected graph will be desirable to accommodate traversal using linked-data query languages so "Version 1" must be reconnected to the other concepts in the model. A **property** is used to create a path between the two attributes to indicate the contribution of "Version 1" to the change's provenance. The path does not show that "Version 1" informs or creates "Attribute 2", while that

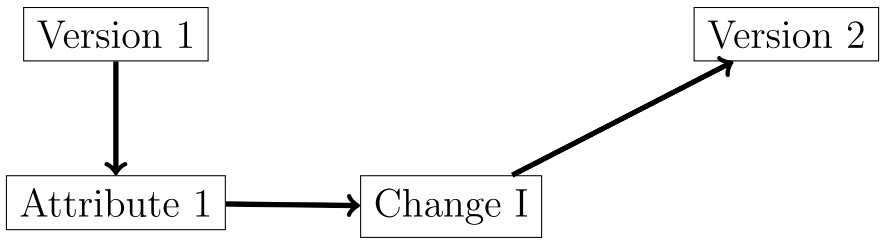

Figure 3.8: Model of the relationships between Versions 1 and 2 when invalidating Attribute 1 from Version 1 as a result of Change I

may be true. The construction was also chosen to create a symmetric orientation with the invalidation change.

3.4.3 Invalidation

The invalidation model has a missing attribute on the right-hand side of the relation, contrary to the addition construction. As a result of the invalidation, an attribute no longer exists in the **right-hand version**. As seen in Figure 3.8, the invalidation change concept matches to the Version 2 object. Just like in the addition model, the invalidation construction maintains a link between the two version objects. In the invalidation case, it makes more conceptual sense, however, because "Version 2" invalidates "Attribute 1" by omitting the attribute.

3.5 Utilized Data Sets

3.5.1 Noble Gas Data set

The "Global Database on ^{3}He/^{4}He in on-shore free-circulated subsurface fluids" is a tumultuous database [92]. The first version, published in June 11, 2013, contains the information from 8 regions of the world united into a single file with 199 columns. The next version of the database, published March 8, 2015, reduces the number of columns to 54, marking a drastic change. In addition, several columns changed the units with which they reported measurements. While usage documentation, explaining the content and use of the data, accompanied each version, no records were included indicating what changed between versions. A change log would be valuable guide with such drastic structural and content changes. The third and most recent publication came in July 11, 2017, with no changes to the number

Table 3.1: Files in the Noble Gas data set.

Filename	File Size (Bytes)	Rows	Columns	Total Cells
DB_HE_6733.xlsx	2682683	6733	199	1339867
DB_final-55-7262_2015_03 _08.xlsx	2729060	7265	54	392310
NG_DB_final_2017_07 _01.xlsx	4216595	8231	54	444474

Table 3.2: Files in the Copper data set.

Filename	File Size (Bytes)	Rows	Columns	Total Cells
ParageneticModeTable_Cu_6. 8.2016.xlsx	339175	705	37	26085
ParageneticModeTable_Cu_8. 21.2016.xlsx	233715	685	51	34935

of files or columns, but many new rows. The structural summary of each of the files can be found in Table 3.1.

3.5.2 Copper Data set

The Paragenetic Mode for Copper Minerals database became available through collaboration with the author's lab to create new methods of visualizing mineralogy relationships [93]. The first version was collected June 8, 2016, with the update following soon after on August 8, 2016. Major edits are fairly limited with only 16 column additions and 2 removals between the versions. Value formats remain consistent from one version to the next, resulting in a much more condensed body of changes, making transitions more easily verifiable. Compared to the Noble Gas data set, it provides a more stable data platform to implement the versioning model in Section 3.3. The data from Copper Minerals is also more processing friendly, making it agreeable to automatic change log generation. An interesting thing to note in Table 3.2 is that the second version takes up less storage space even though it has more data.

3.6 Summary

The versioning model provides a method to capture change information in greater detail than current provenance models. The inclusion of versions and attributes into the model connect changing items with the objects they influence. The changes create a ladder-like structure to connect version objects in greater detail. Each rung of the ladder can not only be counted, but also grouped into types of change according to the respective operation. The method of instantiating a versioning graph will be covered in Chapter 5.

CHAPTER 4
MACHINE-READABLE CHANGE LOG

4.1 Introduction

Change logs explain the differences between versions; however, they are often only available in human-readable formats. Readability puts a limit on the length and extent of the change log since a human will need to write it. Manageable change descriptions become difficult with large data sets featuring many changes, or data sets that change often, but large data sets are exactly the data sets which need change logs the most. Automating the process will allow more data sets to provide change documentation in a timely fashion for data sets. Encoding the change logs with structured data will provide a means for users to efficiently consume change information. The additional encoding will inflate the size of a standard change log which becomes an issue with the change logs.

Change logs were generated for two data sets, the "Global Database on ^{3}He/-^{4}He in on-shore free-circulated subsurface fluids" data set and the Paragenetic Mode for Copper Minerals database. Following the practices of other change logs, the documents present before and after values for comparison which can be seen in Figure 4.1. The change logs identify challenges to adopting thorough change logs as a practice in versioning data sets.

Change Log

Abswurmbachite

Column v1	Column v2	Version 1	Version 2
	9 (12)		0.0
	11 (14)		0.0

Figure 4.1: Abswurmbachite entry in the Copper Dataset Change Log

4.1.1 On the Importance of Change Log Automation

The NASA's Cold Land Processes Field Experiment (CLPX) collects ice measurements from a wide number of sites in the arctic [94]. The data was published originally in ASCII, but the data was later ported into shapefiles as well. A known issue with the transition was that comments in the Summary data set were left out due to format limitations. A change log was created to determine whether additional differences existed within the two formats.

The initial problem with forming a change log is that rows in the data did not include unique identifiers to align attributes within the versions. A unique tuple, using values from four columns of the data as seen in line 72 of Appendix G, was created to match rows together since the order of the rows did not correspond between the two formats. An immediate problem that resulted was some differences occurred within the columns necessary to form the unique tuple, namely some values disagreed by 0.5. Following the correction, the data could be correctly aligned. The change log generating script standardizes the comparison process to encompass every entry in the `generate_logs` function. As seen in lines 7 through 61 in Appendix G, the script can be modularized to handle different shapefile structure while still using the same `generate_logs` function.

The change log found that for columns "soil sample A" (SL_A) and "soil sample B" (SL_B), many entries consistently disagreed between versions. Disagreements resulted from values being recorded with 'y' or 'n' for a number of values but with numbers for the rest. Since the values in the shapefiles are stored in a database format, a column must conform to a pre-specified type, causing the character encodings to appear as 0s in the shapefiles. Additionally, the columns experienced 0.5 misalignments and null values of -999.0 being encoded as 0. The columns are encoded as integers, causing the disagreement.

With further investigation, misalignment in the values caused repeated inquiries into differences between expected and reported values. Parsons et al. [95] state that the, "text files are the primary archival format of the data." The solution resolve differences between the formats was to defer to an official version rather than compute or provide change information. The deference to a different format, unfortunately partially undermines the benefits of increased accessibility provided by the shapefile format.

4.2 Encoding a Change Log

Very little natural language is used in the change log to regularize the format and improve compatibility with RDFa. The change logs follow a common format with three sections as seen in: Additions, Invalidations, and then Modifications. The sections may be further grouped by column or row additions. The division means that changes are not published into the change log as they are found, but instead organized and grouped beforehand.

Employing RDFa means that the document must be written using HTML formatting. Listing 1 shows the text necessary to layout the first four lines of Figure 4.1, showing modifies for the Abswurmbachite mineral from the Copper Minerals database. While the content only shows four lines, the underlying markup takes up three and a half times as many lines. Line 2 of Listing 1 states that all following resources will be attributes of Version 1. Line 3 defines such an attribute. Lines 5 through 8 define the changes Abswurmbachite undergoes. Because RDFa allows the statements to be embedded within the content, the triples can appear along with the text they describe. Lines 11 and 12 define complete triples which do not appear in the visible document. The lines complete the graph, but must be included in HTML span tags because RDFa only allows a single triple within each tag. Modifying the tags' order so that the spans are unnecessary would cause the visible content to appear in a confusing and disorganized order, rendering the document machine-readable but not human-readable.

After encountering the limitations of using RDFa to include the versioning graph into the change log, JSON-LD was used. The JSON data does not need to conform to the structure or ordering of visible content in the change log. Listing 2 provides the alternative encoding of the Abswurmbachite entry from RDFa. The entry is significantly longer, almost three times longer than the RDFa entry and ten times longer than the original visible content. Instead of including all the data in the beginning or end of the document, each change block is separated into the particular

```
<h3>Change Log</h3>
  <div about="Version1" rel="vo:hasAttribute">
    <div resource="v2:Abswurmbachite" typeof="vo:Attribute">
    <span style="font-weight:bold" property="http://www.w3.org/2000/01/rdf-schema#label">Abswurmbachite</span>
    <table rel="vo:Undergoes">
      <tr  about="ChangeAbswurmbachite12" typeof="vo:Change">
        <td align="right" rev="vo:Undergoes" resource="v1:AttributeAbswurmbachite12v1" typeof="vo:Attribute"> 9</td>
        <td property="vo:resultsIn" resource="v2:AttributeAbswurmbachite12v2" typeof="vo:Attribute">(12)</td>
        <td>          </td>
        <td>       0.0</td>
        <span about="Version1" property="vo:hasAttribute" resource="v1:AttributeAbswurmbachite12v1"></span>
        <span about="Version2" property="vo:hasAttribute" resource="v2:AttributeAbswurmbachite12v2"></span>
      </tr>
</table></div></div><br>
```

Listing 1: Abswurmbachite RDFa

div section for that change. This choice allows consumers to extract pertinent change information without needing to ingest the entire versioning graph.

```
<h3>Change Log</h3>
  <div about="v1:Abswurmbachite">
  <span style="font-weight:bold" property="http://www.w3.org/2000/01/rdf-schema#label">Abswurmbachite</span>
  <table>
    <tr  id="ModifyChangeAbswurmbachite12">
      <td align="right"> 9</td>
      <td >(12)</td>
      <td>          </td>
      <td>       0.0</td>
```

```
      <script type="application/ld+json">
[
  {
    "@context": "https://orion.tw.rpi.edu/~blee/provdist/GCMD/VO.jsonld",
    "@id": "http://CUdb.com/v1/AttributeAbswurmbachite9",
    "@reverse": {
      "hasAttribute": "Version1"
    },
    "@type": "vo:Attribute",
    "label": "Primary",
    "undergoes": "http://orion.tw.rpi.edu/~blee/provdist/CU/DTDI/CUjsonlog.html#ModifyChangeAbswurmbachite12"
  },
  {
    "@context": "https://orion.tw.rpi.edu/~blee/provdist/GCMD/VO.jsonld",
    "@id": "http://orion.tw.rpi.edu/~blee/provdist/CU/DTDI/CUjsonlog.html#ModifyChangeAbswurmbachite12",
    "@type": "vo:ModifyChange",
    "resultsIn": "http://CUdb.com/v2/AttributeAbswurmbachite12"
  },
  {
    "@context": "https://orion.tw.rpi.edu/~blee/provdist/GCMD/VO.jsonld",
    "@id": "http://CUdb.com/v2/AttributeAbswurmbachite12",
    "@reverse": {
      "hasAttribute": "Version2"
    },
    "@type": "vo:Attribute",
```

```
    "label": "Primary"
  }
]
      </script>
    </tr>
</table></div><br>
```

Listing 2: Abswurmbachite JSON-LD

The change logs created with RDFa or JSON-LD demonstrates progress towards documents which are both human and machine-readable. The implementation provides evidence that JSON-LD is better suited to embed a versioning graph into a change log than RDFa. RDFa suffers limitations since it is constrained by the content's structure. The modification relation presented in Figure 5.1 is unbalanced and the right-hand side of "ChangeCAM00111" links only to the column attribute but not to the corresponding row attribute. This stems from a mismatch between the model's structure, the order in which data appears in the change log, and the way RDFa links properties together. Because the row label forms the outermost encapsulation, it cannot instantiate both row identifiers and implicitly link them separately. To do so would require explicitly instantiating the attribute in a non-visible part of the document, defeating the purpose of using RDFa to implicitly encode the versioning graph into the document.

Both structured data implementations break up the graph across attributes so that individual parts of the graph can be extracted. The practice of a one-node JSON object is generally helpful for many web applications to load data quickly, but since the change log is not an application, it makes more sense to break up the content. Changes to individual attributes can be identified using anchors on the web page, then agents need only extract and parse the linked-data to the attributes' specific entries. This way, a subgraph of only the pertinent attributes can be created without first ingesting the entire versioning graph.

An unexpected challenge with the change logs is the larger file size and difficulties in loading the Noble Gas data set's JSON-LD change log. The problem

Table 4.1: Noble Gas change log size: 1st Transition

Encoding Type	File Size (Bytes)	% of File 1	% of File 2
Text	5575405	207.8294	204.2976
RDFa	62175478	2317.660	2278.2745
Turtle	80919783	3016.375	2965.1156
JSON-LD	130134071	4850.893	4768.4577

results from needing ten lines to express a single row in the change log. Noble Gas also had an impressive number of modifies, some of which are shared across all rows in the data set. Repeated modifies over rows would account for the explosion in entries within the change log.

4.3 Change Log Analysis

With a trade-off of 14 HTML lines for every visible line and 40 HTML code lines to each visible line, space utilization is a very present concern. Table 4.1 shows the size of each encoding of the change log as well as the percentage in size as compared to either of the files involved in the version transition. 'Text' denotes the encoding control where no structured data is included into the change log. Alone, the control is already double the size of either file. The RDFa encoding more than 20 times the size of the original files, exceeding the size of the control by more than ten fold. A separate file was generated in turtle format to observe whether taking just the linked-data values would reduce the information to a more manageable size, but the turtle file was still 30 times the size of the original files. All file types can be independently generated using the modularized script in Appendix A. In the `write_removed` function on line 184, the text consumption of each data format can be compared by looking at the text written in each mode. Adopting the versioning model and encoding it into a change log will very likely require significant storage investment.

Table 4.2 shows the change log sizes for the second version transition in the Noble Gas data set. Notice that the second transition has much smaller text encodings compared to the original files. The RDFa and JSON-LD encodings once again 10 and 20 times, respectively, the size of the text encoding. The turtle encodings,

Table 4.2: Noble Gas change log size: 2nd Transition

Encoding Type	File Size (Bytes)	% of File 2	% of File 3
Text	403227	14.7753	9.56286
RDFa	4168390	152.7409	98.85678
Turtle	4515435	165.4575	107.0872
JSON-LD	8095372	296.6359	191.9884

Table 4.3: Noble Gas Turtle files

Filename	Add	Invalidate	Modify	Total Triples
changelog.ttl	608	216	102830	110602
changelog2_3.ttl	990	24	5369	8146

representing the most raw form of data without additional HTML text, surprisingly ends up larger than the other data encodings. The explanation is that in order to automate printing the formal triple statements, fewer elements can be smartly implied, leading to larger file sizes even though no markup symbols are in the file. Looking at Table 4.3, the second transition had 20 times fewer modification entries, leading to a much smaller turtle file.

Another way to evaluate the performance of the change log is to look at the number of change entries compared to the number of changed values in the Copper database's case spreadsheet cells. From Table 4.4, the behavior of the encodings is very similar to the second transition of Noble Gas. The text format is smaller than the original data set, but the encoded files are at least 10 times the size of the database files. To determine the number of cells affected by a change, the number of cells added by new rows is summed with the number of cells added by new columns, using the width and length of Version 2. The cells affected by removals is based on the length and width of Version 1. The number of remaining cells, equivalent

Table 4.4: Copper change log size: 1st Transition

Encoding Type	File Size (Bytes)	% of File 1	% of File 2
Text	140131	41.3152	59.9580
RDFa	2032823	599.343	869.787
Turtle	1538772	453.680	658.396
JSON-LD	3500067	1031.93	1497.57

Table 4.5: Changes to Copper Data

Change Type	Rows	Columns	Cells Affected
Add	1	16	10995
Invalidate	21	2	2145
Modify	NA	NA	2628

Table 4.6: Change capture compression in Copper Data

Change Type	Triples	% of Cells Affected
Add	17	0.065%
Invalidate	23	1.1%
Modify	2628	100%

between the two files is 23940. Since modifies are reported cell-by-cell, the number of cells affected is equal to the number of modifies, 2628. The rows and columns that modification affects are not available because the changes appear inconsistently across the rows and columns meaning a reported value would be misleading. The complete counts are reported in Table 4.5.

The triples used to explain changes as a percentage of the cells affected is reported in Table 4.6. Smaller percentages indicate how well one triple can explain changes to multiple cells by compressing the number of entries. Notice that additions are much more efficient in explaining changes than invalidations due to triple explains a row or column relation. Invalidations explained changes to rows primarily while additions mostly explained changes to columns, but since columns are much longer, additions ended up scoring higher on efficiency. Modification triples do not compress change information well and also account for more than a majority of the changes to the data, meaning that modification triples most likely account for the bloat in the physical representation of the triples. Not represented in the change log are the unmodified cells which account for 89.02% of the matching cells between the Copper files. The analysis indicates that while addition and invalidation may be very efficient in expressing changes, improvements to encoding and modification capture are needed to bring down the storage costs of automated change logs. The bloated change log size likely explains the dearth of data set change logs in practice since using the storage space for more data would be more valuable.

4.4 Summary

The automated change log generation yielded some unexpected results. Automated change logs standardize the process to capture change within a data set. While more popular text-only change logs could be adopted, a versioning data model was necessary to make the logs also machine computable. The computability improves digital comprehension of the change document. The drawback is that the encoded change logs are reliably much larger than the original data set in bytes. The storage space cost likely contributes to the reason that change logs are often unseen in data set documentation. The automation and inclusion of change logs inform consumers how much the data set has changed.

The human-readable presentation defines the structure which tags in the change log must take since maintaining human-readability is desired. The structure then determines the order in which linked-data statements must appear in the log to encode the change log using RDFa. The ordering creates limitations on how strictly the encoded graph adheres to the specification from Chapter 3. While construction of the change log is automated, encoding through RDFa significantly reduces the source HTML readability. In other applications using RDFa, the triples describe and link the text encapsulated by HTML tags. The versioning graph exclusively ignores the marked up content and links together tags or explicitly defines full triples in span tags.

Change logs are much less restricted when encoded using JSON-LD rather than RDFa. The encoding format pulls the graph out of the attributes where they do not interact with content and into a separate script section. The method causes a drastic expansion per change in necessary text. The decision to divide up JSON-LD objects by the row in the change log they describe likely contributes significantly to the overhead necessary for the encoding. The division was made with the forethought that change log consumers may desire to only ingest specific subgraphs of the versioning graph. Separated JSON-LD objects will likely need to be merged in the future to save space for data sets with many changes.

The resulting logs end up very large and sometimes do not load in a browser. Reassuringly, both data sets displayed the same space usage complexity with RDFa being ten times the plain text size and JSON-LD twice the size of a change log in RDFa. The relationship unfortunately means that a JSON-LD change log, with more readable source code, is twenty times the size of a plain text change log. In order to retain usability, there will need to be methods optimizing change log structure or representation.

CHAPTER 5
CHANGE METRICS

5.1 Introduction

Machine computable change logs provide a very powerful means to begin answering basic versioning questions in a formal and systematic manner. From the change log, a linked-data versioning graph can be extracted and the changes counted to communicate how different are two versions. The data model was constructed to allow a wide variety of ways to connect together versions such that more complex analytics could be performed using the versioning graphs. The analytics show that producers must be very transparent when communicating the methods data producers use to assess change impacts as shown in Section 5.4.2.

When versioning a data set, researchers very rarely ask whether two objects can be compared. The data producer often establishes the context in which data objects are sufficiently similar—to use terms from FRBR—**expressions** of the same **work**. Confirming the context prior to making version comparisons is fundamental to ensuring that the resulting versioning graph contains meaningful results. The data sets described in the following section have sufficient context as established by their producers. Using the data in these data sets, the model from Chapter 3 is instantiated into versioning graphs. The versioning graphs are encoded into HTML change logs using RDFa and JSON-LD. These versioning graphs allow for an analysis of the change between versions, which gives insight into the version identifier. Finally, a versioning graph is used to classify the kinds of change separating versions of a data set to determine the utility.

5.2 Implementing the Versioning Model

The following subsections detail the steps used to implement a versioning graph using the model defined in Chapter 3 and the challenges encountered. Section 5.2.1 discusses the decisions made to align the attributes within the Noble Gas dataset and within the Copper data set. The alignments create a formula to detect changes and assign them to either an addition, invalidation, or modification change. A change log can then use the assignments to organize a presentation of the change data. The underlying versioning graph exists as linked-data encoded within the change log, but can also appear as explicit linked-data statements. The linked-data uses VersOn, a versioning ontology made by me, to express the data model using the *vo:* namespace. The procedure within this section defines the process used to create versioning graphs found in all the following sections of this Chapter.

5.2.1 Form a Mapping

A mapping specifies the method to determine the attributes of a versioning graph and how to compare them. For spreadsheets and table-based data, row and column indexes initially seem an ideal attribute, but edits often show the contrary. The Noble Gas data set needed a mapping to align the spreadsheet's columns since 140 columns were removed from Version 1. The remaining columns in the Version 2 no longer had the same column indexes that they did in Version 1 so the column headers were used instead. The Copper data set retains many of the original columns, but their ordering has changed between versions. In addition, rows must be aligned since both a row and column attribute are necessary to uniquely identify a cell. The Noble Gas data set split up its rows across eight files, each file representing a separate region of the Earth. Cells need to be uniquely identified since this is where a comparison will be made to determine whether a modification change has occurred in a spreadsheet.

Once aligned determining which attributes have been added, invalidated, or modified is straightforward. Attributes which only exist in the original or left-hand version have been invalidated. More specifically, a set of attributes

$$\mathcal{I} = \mathcal{R}_l - \mathcal{R}_r$$

where $\mathcal{R}_l$ and $\mathcal{R}_r$ correspond to the row identifiers of the left-hand and right-hand versions, respectively. Likewise, a set of attributes

$$\mathcal{A} = \mathcal{R}_r - \mathcal{R}_l$$

contain all the added attributes. Performing the same operations on the columns result in sets of the added and invalidated columns. A script then iterates over the remaining cells which exist in both versions to determine if they differ, resulting in a modification change. The unchanged cells form a set of entries which do not appear in a change log or the versioning graph. The attributes in these sets are then minted into URIs and linked together into the versioning graph, or they can be used to publish a change log.

5.2.2 Generate Versioning Graph

The versioning graphs presented in this section were created by extracting triples, using the code in Appendix E from the associated change log which was covered in Chapter 4. The `extracting` function of the script extracts the JSON objects in every script tag, enters the values into an RDF graph, and then prints the graph out to a file. The script has been generalized to process every JSON-LD encoded change log made in this thesis. The statements making up the versioning graph could have alternately been published by writing out the triples directly instead of encoding them into a change log. Figure 5.1 displays a subgraph of the Noble Gas data set's versioning graph between Versions 1 and 2, highlighting each of the AIM changes. Notice how the versioning graph differs from the provenance graph in Figure 5.2. The versioning graph unpacks the *prov:wasRevisionOf* relationship into explicit components. These components reveal more detailed differences between version 1 and 2 of CAM001 in the provenance graph which are the differing compilation activities. The change log encoded the triples in RDFa, resulting in the attribute "AttributeCAM00111v2" to the right of the modification change. Because RDFa does not naturally support multiple predicates while also conforming to the content structure of the change log, an attribute was created to combine both the row and column identifier for the changing cell. Separating the attributes would require multiple dedicated HTML tags which don't appear along with content. Including these tags would diverge from benefits of encoding triples as attributes. Figure 5.1 also shows that even though many columns are added when a new row is added, the row identifier can be used to summarize the columns additions.

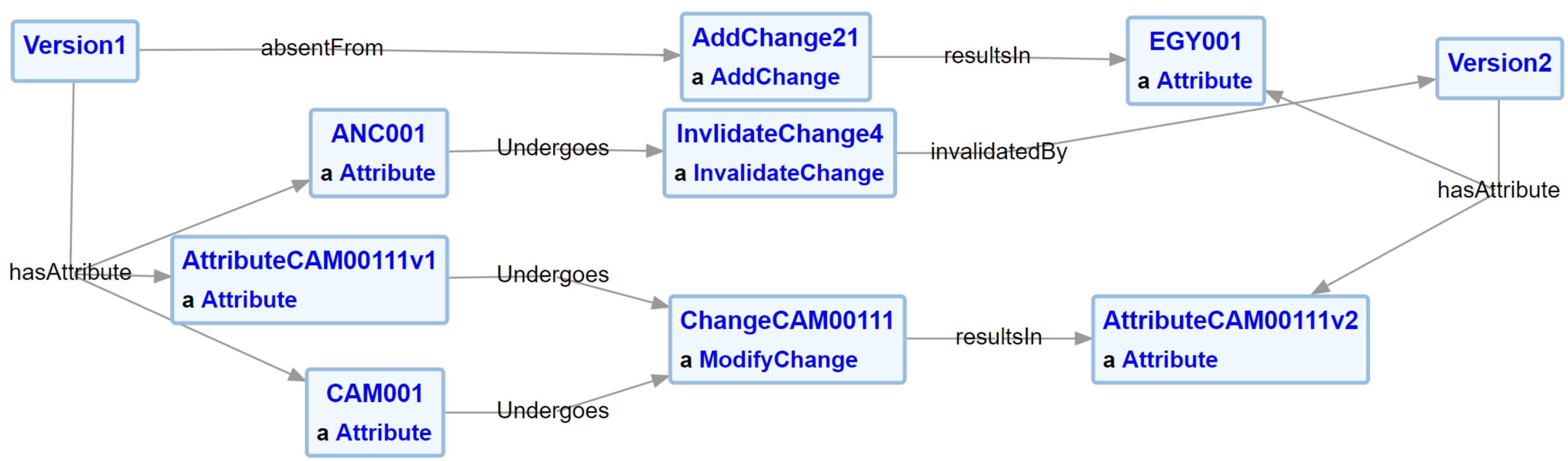

Figure 5.1: Some initial entries from versions 1 and 2 of the Noble Gas data set

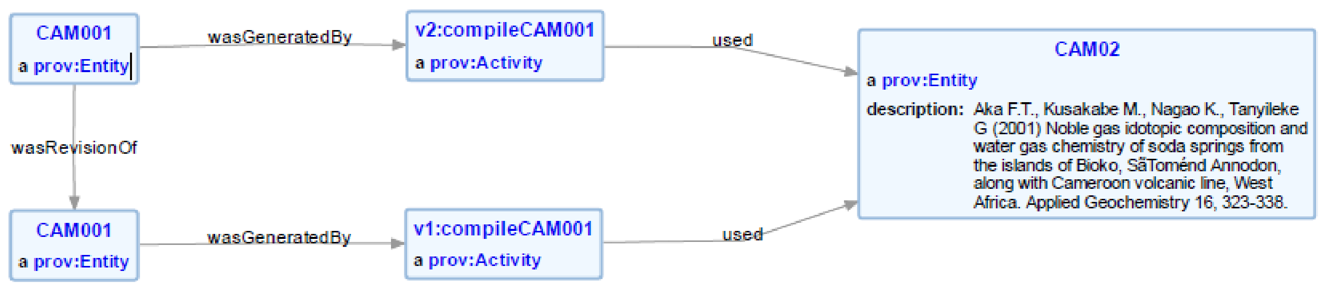

Figure 5.2: Provenance graph for the CAM001 entry of the Noble Gas Database. Other than the labels, the structure of each data object is very much the same.

Another modification to the implementation differs from the original versioning model. The modification construction defined in the model only covers the case where a single attribute is sufficient to define a change. The modification captured in spreadsheets describes a cell which requires a row and column identifier to indicate uniquely. The implementation demonstrates that using multiple attributes is an allowable, sometimes necessary, construction.

Listing 3 presents the statements in turtle format necessary to express that the entry EGY001 has been added to the data set from Version 1 to Version 2 as shown along the top of Figure 5.1. The namespace for many of the URIs is ⟨http://rdfa.info/play/⟩. RDFa allows identifiers to refer to an element on the web page, and the web tool which generated the triples from RDFa, therefore, used its URL as a namespace to produce a valid URI.

```
<http://rdfa.info/play/Version1> a vo:Version ;
vo:absentFrom <http://rdfa.info/play/AddChange21> .
<http://rdfa.info/play/AddChange21> a <https://orion.tw.rpi.edu/~blee/VersionOntology.owl#AddChange> ;
vo:resultsIn <http://rdfa.info/play/Attribute21> .
<http://rdfa.info/play/Attribute21> a <https://orion.tw.rpi.edu/~blee/VersionOntology.owl#Attribute> ;
rdfs:label "EGY001"
<http://rdfa.info/play/Version2> a vo:Version ;
vo:hasAttribute <http://rdfa.info/play/Attribute21>
```

Listing 3: Noble Gas Statements in Turtle for Addition 21 as an example for instantiating a versioning relation

Figure 5.3 shows a similar subgraph from the Copper data set versioning graph. The versioning graph was assembled using an RDFa change log and also displays a merged attribute on the right side of the modification. In the full versioning graph, multiple of each change is present, forming a zipper or ladder-like structure. As a result, each addition, invalidation, or modification is given separate names for each instantiation.

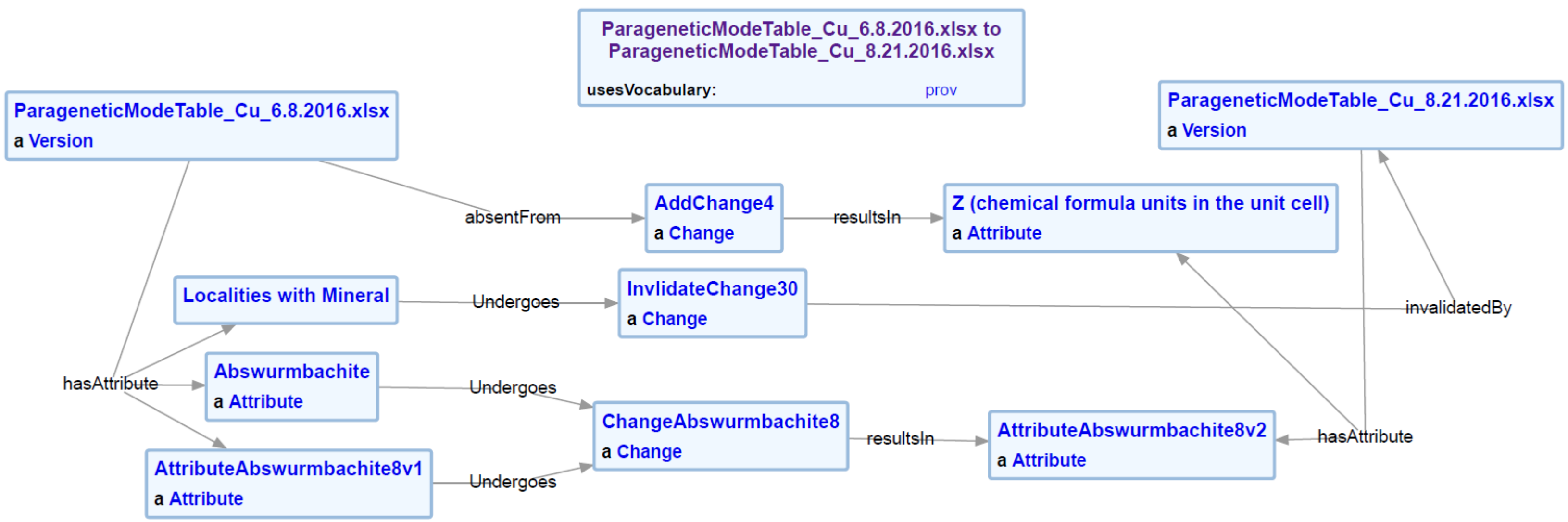

Figure 5.3: Versioning Graph representing the Copper Minerals linked data graph with selected entries of additions, invalidations, and modifications.

5.2.3 Graphs with Multiple Versions

Figures 5.1 and 5.3 depict a comparison between only two versions, but a project can contain more than two objects. Case in point, Version 3 of the Noble Gas data set was released on July 11, 2017. Figure 5.4 shows a subgraph that contains changes from all three versions of the Noble Gas data set. From Version 1 to Version 2 of the data, EGY001 becomes introduced as an attribute into the data set. This entry then undergoes a modification in columns 29, 31, and 43 when comparing Version 2 and Version 3. Entry TUR030 goes through a modification change in column 11 from Version 1 to Version 2. The entire row, however, becomes invalidated in Version 3.

Notice the difference in how Figure 5.1 and Figure 5.4 refer to columns. Figure 5.1 used linked-data extracted from a change log employing RDFa, forcing the row identifier and the column identifier into the same concept. The way nesting works in RDFa means that ChangeCAM00111 cannot back reference multiple concepts in a single statement, therefore AttributeCAM00111v2 was used to imply CAM001. Figure 5.4 used linked-data extracted from a JSON-LD encoded change log. Since the change log can use explicit statements, the column identifier refers to the entire column and can be used to identify changes in the same column across multiple rows.

5.3 Change Metric

Research Question 2 seeks to determine the extent of differences between two versions. Many versioning systems use dot-decimal identifiers to signify whether a change is large, medium, or small. The exact requirements to determine change size differs widely across different domains and applications. The versioning graph provides a new, more regular method to quantify change between objects using versioning operations. The work done with Global Change Master Directory (GCMD) Keywords shows the qualitative relationship between version identifiers and change distance. Work with the Marine Biodiversity Virtual Laboratory (MBVL) data set then extends VersOn to give more detailed accounting with the change capture method.

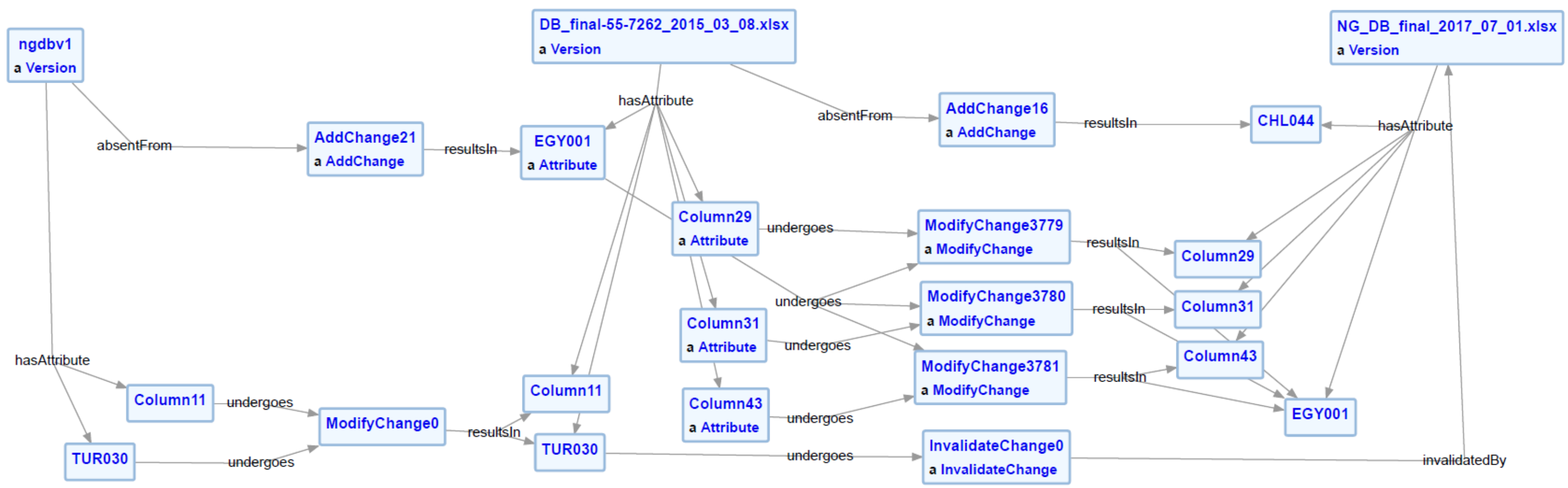

Figure 5.4: Versioning Graph representing the Noble Gas linked data graph with selected entries of additions, invalidations, and modifications after the publication of the third version.

5.3.1 Utilized Data Sets

5.3.1.1 Global Change Master Directory Keywords

The GCMD is a metadata repository used by NASA to store records of its available data sets [96]. They employ a set of keywords to make NASA Earth Science data sets searchable. These words tag and label datasets into strictly defined categories [97]. GCMD Keywords do not qualify as a standard web ontology since GCMD Keywords do not constitute a class hierarchy. The management team stored early versions of the keywords in Excel spreadsheets, later using a centralized distribution system, but data is not available prior to June 12, 2012. The Key Management Service (KMS) now serves the keywords directly in a variety of formats. Each version of the keywords, encoded in Resource Description Framework (RDF), was downloaded into separate files. Only versions from June 12, 2012 and after were available, resulting in 9 version files. Each keyword corresponds to a unique identifier, and when combined with a web namespace, resolves to a data description of the keyword. Every identifier can be referred to per version by including the version's number at the web identifier's end, meaning that identifiers are consistent across versions. The taxonomy uses the concepts *skos:Broader* and *skos:Narrower*, where skos refers to the Simple Knowledge Organization System (SKOS) ontology name space, to form a tree hierarchy [98]. The tree's root is the keyword, "Science Keywords." The data set provides an interesting study case due its long sequence of versions and ready use of linked-data technology [99].

5.3.1.2 Marine Biodiversity Virtual Laboratory Classifications

The MBVL, based at Woods Hole Oceanographic Institution, provides data and services for the study of marine biology with an integrative approach [100]. In the application studied, a choice of algorithm and taxonomy pairings must be tested on a known population in order to estimate their performance with an unknown microbial population. The original sequences belong only to the species listed in Table 5.1. The original population's census is not available to the author, and only the list of species is known, forming the first data set in this section. These sequences are then grouped and classified by a specific taxonomy and algorithm

Table 5.1: List of species in the original population.

Acinetobacter baumannii	Actinomyces odontolyticus	Bacillus cereus
Bacteroides vulgatus	Clostridium beijerinckii	Deinococcus radiodurans
Enterococcus faecalis	Escherichia coli	Helicobacter pylori
Lactobacillus gasseri	Listeria monocytogenes	Neisseria meningitidis
Porphyromonas gingivalis	Propionibacterium acnes	Pseudomonas aeruginosa
Rhodobacter sphaeroides	Staphylococcus aureus	Staphylococcus epidermidis
Streptococcus agalactiae	Streptococcus mutans	Streptococcus pneumoniae

pairing. The workflow utilizes two taxonomies, the Ribosomal Database Project (RDP) and the Silva taxonomy. Using these databases, the Species-level IdentificatioN of metaGenOmic amplicons (SPINGO) or the Global Alignment for Sequence Taxonomy (GAST) algorithms assign taxonomic ranks to each sequence. The process produces four data sets, each using the same grouping identifiers and having the same size in each group. Since the data sets have the same number of sequences, the primary differences between the data sets are the ranks assigned to each sequence.

5.4 Global Change Master Directory

5.4.1 Global Change Master Directory Versioning Graph

The GCMD establishes the context that each **manifestation** of their keyword list is related versions. Since the unique identifier for each keyword remains the same across versions, the unique keyword identifier can be used to align a mapping across versions. Additions and invalidations are detected by checking an identifier's presence within both versions. Notice that the presence can be computed, as shown in lines 22 and 23 of C, without iterating through the data set. A modification occurs when a keyword's *skos:Broader* property differs between adjacent version. The alignment assumes that there is not reason a keyword's preferred label would change, but still reports a value when it has new entries in the "notes" property. A difference indicates that the word has been moved to a different place within the taxonomy since identifiers do not change across versions and a keyword only has one parent concept. Changes over consecutive versions can be collected into a single graph using the method in Section 5.2.3 to chain together versioning graphs. A change log was generated for each pair of consecutive versions in GCMD Keywords and embedded with JSON-LD. Versioning graphs for each adjacent version was created by extracting JSON-LD from the corresponding change log, and entering the triples into a Fuseki triple store.

5.4.2 Connecting Change Counts to Identifiers

The addition, invalidation, and modification counts for each transition are presented in Figure 5.5. Modify changes are labeled as Moves to differentiate between types of modifications when discussing Version 8.5. The query used to extract the counts is found in Listing 4. Notice the sharp spike in additions and invalidations when transitioning from Version 8.4.1 to 8.5. The version identifiers indicate that at most a minor or technical change has occurred, but the counts of addition and invalidation in this transition is more than triple the counts in either of the previous **major** transitions. Not only should a small transition not produce changes of this quantity, but the data set's size is on the order magnitude of the recorded invalidations. In addition, no modifies are revealed, and even the root node "Science Keywords" has been invalidated. Further investigation of the root word reveals that the name space for the keywords has changed from `http` to `https`. To provide context, NASA mandated a transition to secure protocols, and the group changed the name space to ensure the URIs remained resolvable. Since the identifiers are unique, the new name space means they no longer refer to the same object after the protocol change. Because the keyword identifiers no longer match, the mapping approach results in the total invalidation of keywords from 8.4.1 and the addition of keywords from 8.5. The dot decimal identifier for the transition from Version 8.4.1 to 8.5 does not match the number of changes in the versioning graph.

Changing the mapping method to account for the new namespace provides a pathway to compare the perceived change by the producer as evidenced by the version identifier with the change distance in the versioning graph. To do this, the mapping treats identifiers with `http` and `https` the same as seen in lines 25 and 29 of Appendix D. The code breaks apart the Universally Unique Identifier (UUID) from the URI and uses the prior namespace or current namespace, respectively. Differences in change distance become much clearer after controlling for the

Table 5.2: Global Change Master Directory Keyword Change Counts

Transition	Add	Invalidate	Modify	Total
June 12, 2012 to 7.0	310	9	22	341
7.0 to 8.0	503	6	79	588
8.0 to 8.1	277	28	22	327
8.1 to 8.2	53	1	26	80
8.2 to 8.3	58	0	13	71
8.3 to 8.4	53	0	1	54
8.4 to 8.4.1	86	13	8	107

altered name space in Figure 5.6. All revisions are dominated by additions, but major version changes have counts around 300 to 500 while minor revisions are an order of magnitude smaller. The transition from Version 8.4.1 to 8.5 also seems to follow this trend. The additions in “8.4 to 8.4.1” in Figure 5.6 numbers almost a hundred, providing evidence that the trend of decreasing order of magnitudes may now continue as the granularity of the version identifier increases.

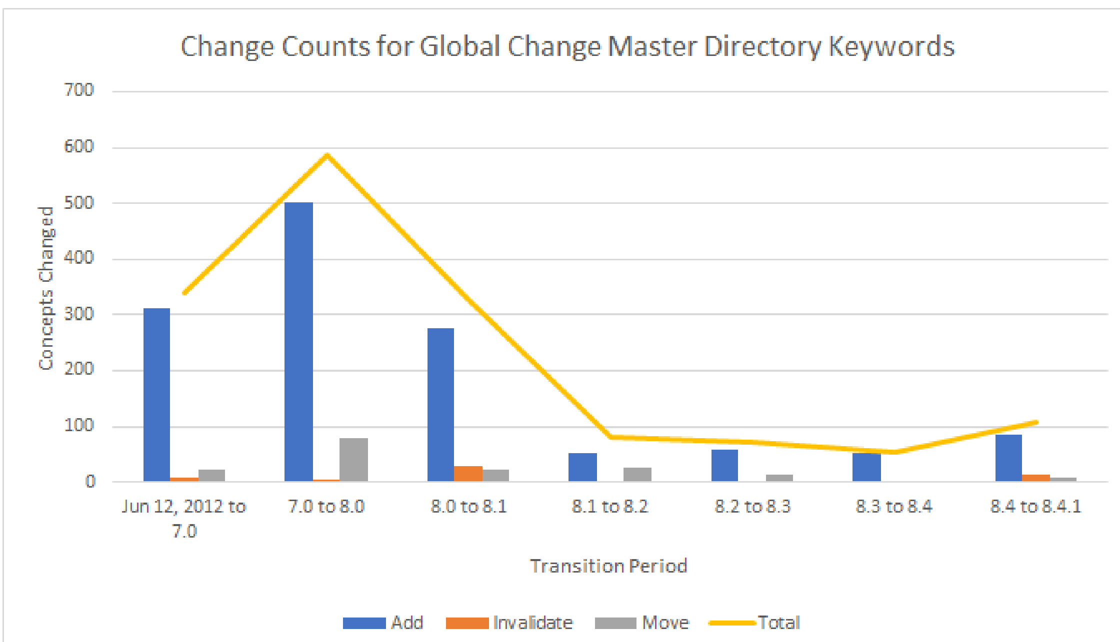

Figure 5.5: Add, Invalidate, and Modify counts from the beginning of the Keyword Management System to Version 8.4.1.

```
PREFIX vo:<http://orion.tw.rpi.edu/~blee/VersionOntology.owl>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

SELECT ?p (COUNT (DISTINCT ?s) as ?count)
{
?s a ?p .
?p rdfs:subClassOf vo:Change .
} GROUP BY ?p
```

Listing 4: Query to compile the counts for each subclass of Change in a GCMD versioning graph.

Table 5.3: Difference in Version 8.5 mapping methods

Mapping Method	Add	Invalidate	Move	Modify
Standard	3097	3031	0	0
Silent	68	2	22	0
Bridged	68	2	22	3007

5.5 Marine Biodiversity Virtual Laboratory

5.5.1 Variant Versioning Graph

The experiment conducts activity over two phases in this procedure. The first phase takes sequences from the original known population and feeds the sequences though a particular algorithm/taxonomy combination to produce a candidate classification. Since the classifications for the known population sequences are unavailable, there is not sufficient context to perform a valid comparison with the candidate classifications. The second phase compares the performances of each candidate classification of a algorithm/taxonomy pair. The use of addition, invalidation, and modification varies slightly in this application since all the results use the same sequences. A versioning graph utilizing just the sequence identifiers would only result in **modify** changes when taxonomic ranks differ since the sequence identifier exists in both data sets. The mapping instead uses the sequence identifiers to align comparisons and then the taxonomic rank classification to determine the kind of change. If the right-hand result specifies more taxonomic ranks, the relationship is an addition, lines 200, 204, 216, and 226 in Appendix F. If the left-hand result is more specific, then the relationship is classified as an invalidation. If both results

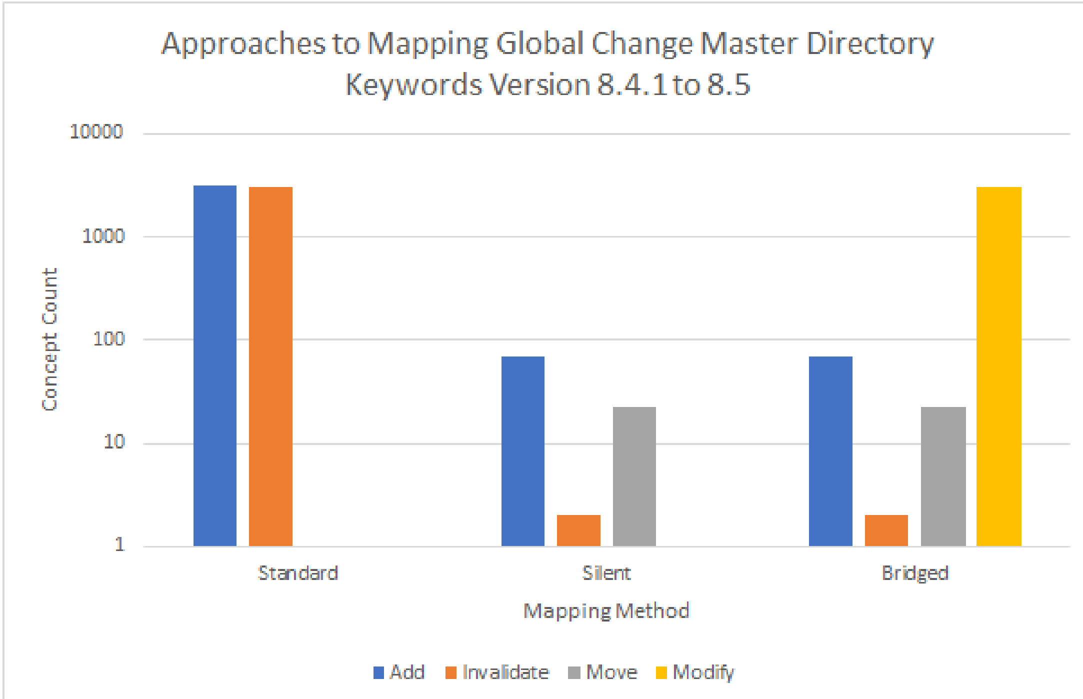

Figure 5.6: Add, Invalidate, and Modify counts using different methods of mapping identifiers in Global Change Master Directory Keywords Version 8.4.1 to 8.5.

have the same precision but the name differs, then the link is a modification. Otherwise, no change is detected. The **invalidations** and **modifications** were marked with '---' and '>>>' respectively in the function `get_output` of Appendix F.

Figure 5.7 shows the changes detected when varying either the taxonomy or the classification algorithm. Only the taxonomy or only the classifier differs in each comparison to control for overlapping influences that having both a different taxonomy and classifier may introduce. Each bar indicates the total number of differences between sequences for a specific kind of change. The bars are further broken down by the taxonomic rank at which the difference occurred. For example, in "Silva vs RDP, Gast", a notable number of classifications differed at the species rank. The graph also indicates that using the RDP taxonomy often produces more precise classifications since both "Silva vs RDP, Spingo" and "Silva vs RDP, Gast" feature a larger number of additions than any other change. The classifier comparisons feature a high number of invalidations; however, "RDP, Spingo vs Gast" also displays a higher number of modifies than invalidations.

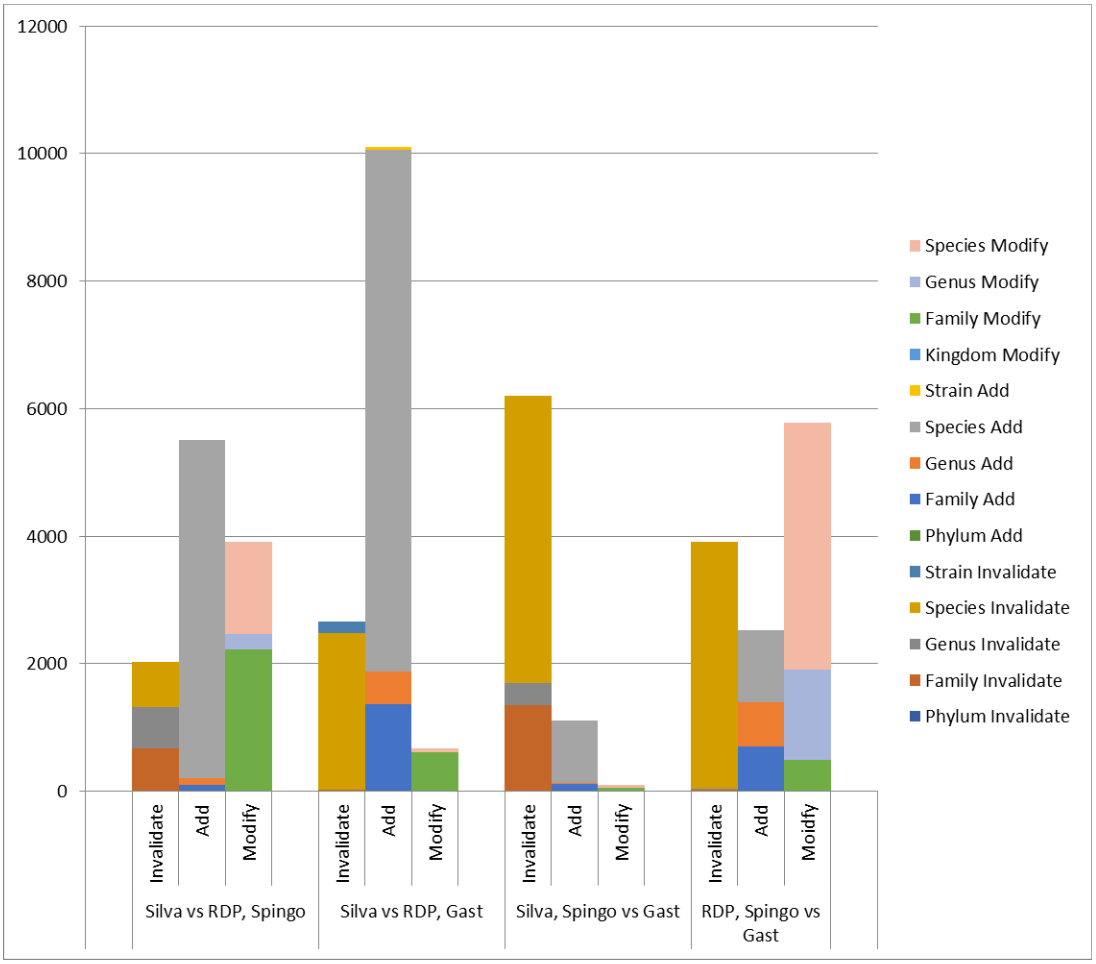

Figure 5.7: Compiled counts of **adds**, **invalidates**, and **modifies** grouped by taxonomic rank across algorithm and taxonomy combinations.

5.6 Version Graph Discussion

The versioning graph successfully addresses the concerns of Research Question 1 by capturing all the differences within the Noble Gas data set and within the Copper data set into a versioning graph. Some additional concerns had to be addressed, such as dual attribute identification, during the implementation of the versioning model. The multiple files in the first version of the Noble Gas data set needed to be collected into a single concept in order to preserve the one-to-one relation between versions. The grouping simplifies the versioning graph structure as well as reduce the complexity of a change log encoding.

In Chapter 3, there is only one attribute on each side of the interaction. Figure 5.3, however, shows two attributes used to characterize the *vo:ModifyChange*. While the model only shows one attribute, it was found that in some applications, multiple attributes may be necessary to properly model a single change. The construction does not even need to have the same number attributes on both sides of the change. The flexibility becomes important when trying to model, for example, a single location entry being split into separate latitude and longitude entries.

The versioning graph's construction allows multiple versions to be linked together. The versioning graph provides not only greater continuity than Schema.org's properties, but also greater detail than PROV-O's versioning properties. Continuity is important since many versioning linked-data alternatives view version change as a single contained *activity*. When linking together multiple versions using a versioning graph, the relationship between non-adjacent versions becomes implied in the graph's structure. The natural pathway between attributes in non-adjacent versions holistically considers the relationships among all attributes along that path. In comparison, other models only capture activity between the adjacent versions.

The model struggles with discontinuous changes to an attribute across multiple versions. Since the model does not capture when an attribute doesn't change, it is possible for an attribute in an earlier version to become disconnected from later versions due to inactivity. For example, in Figure 5.4, column 31 of EGY001 becomes modified transitioning into the Version 3. If that column underwent no activity in the next transition but changed from Version 4 to 5, the connection between all the column 31s would no longer be continuous. This poses a problem for executing queries in a triple store which rely on graph traversals, but no path exists between disconnected attributes.

5.6.1 Version Identification

The versioning process discovered a discrepancy in the identifier assignment in the GCMD Keywords taxonomy. The original analysis was intended to determine if dot-decimal identifiers would reflect an order of magnitude division among the change counts of the versioning graph. According to the *Keyword Governance and Community Guide Document* [101], "Full GCMD keywords list releases get a new major version number (e.g., 8.0). Incremental releases for updates to topics, terms, and variables get a new minor version number (e.g., 8.1). The change counts for Versions 'June 12, 2012' to 8.4.1 demonstrates a threshold of changes necessitating a full keywords list release. The document does not explain the purpose or distinguishing qualities of versions with a new revision number, 8.4.1. Version 8.5, however, was named with respect to perceived taxonomy changes and did not consider underlying linked-data practice revisions. The conclusion can be obtained by looking at 5.6 and noticing that the only matching approach without a bar equal to the size of the entire data set is the 'Silent' approach. For a purely URI based comparison in 'Standard', Version 8.5 definitely falls under the category of full release since an entirely new list of words is released. Users using the 'Bridged' approach would also see a new full release because all old words have had the URIs edited. Version name assignment based on producer perception and not allowing users to assess change measures is concerning. Making sure that data consumers have the ability to assess change in data sets when the requirements for change differs between producer and consumer must be addressed.

The analysis does not to claim that change distance should be the sole mechanism in determining version identifiers. The counts, however, can provide a more quantitative method to compare version differences. In Figure 5.5, the yellow line indicates the total changes made to the data set, performing a similar function as the major/minor/revision version identifier. Breaking up the changes into types reveals additions dominate manipulations to the data set. Addition, invalidation, and modification provides deeper insight into how a data set is changing, but some changes can be more impactful than others which this model does not capture.

5.6.2 MBVL Analysis

In Section 5.3, the versioning process was used to compare the performance of different taxonomy and algorithm combinations. The data set diverges from many of the common understandings of versions since each of the versions are not sequential and are largely independent. The data set of species names in the initial population would not have produced very meaningful results if applied to the versioning model since it lacked sufficient data to map the other data sets together well.

In Figure 5.7, the first set of columns in the Silva taxonomy results are versioned against RDP using the SPINGO algorithm. The naming reflects the orientation in the versioning graph so Silva forms the left-hand version and RDP would be the right-hand version. In this comparison, using the RDP taxonomy seems to provide more accurate results, most specifically at the species level. The taxonomies also disagree fairly often at the species and family ranks. Switching to the GAST algorithm in the second set of columns, RDP once again demonstrates a noticeably greater accuracy in species classification. There are also significantly fewer disagreements using the GAST algorithm between the two taxonomies. Looking at the third set of columns, Silva demonstrates greater accuracy classifications under the SPINGO algorithm than under GAST. Over four thousand of these entries can be classified to the species level when GAST cannot. In the fourth set of columns, RDP appears to perform better with SPINGO than GAST. However, the comparison is dominated by a much larger number of disagreements between almost six thousand entries, primarily at the species rank. On closer inspection, this disagreement is explained by GAST classifying the species for a number of entries as “uncultured bacterium”. This analysis presents evidence that using the RDP taxonomy with the SPINGO algorithm will produce the most accurate classification results.

5.7 Summary

The results in this chapter implements the versioning model and demonstrates the process and challenges experienced in this endeavor. The entries in a data set is separated into groups of additions, invalidations, modifies, and unmodified by their attributes. These operational groups organizes the data into a form to publish into a versioning graph. The approach used to create the versioning graph involves extracting the linked-data from a marked up change log. The decision resulted in constrained representations of the versioning graph, resulting from demands of the encoding methods. Versioning graphs created using freer form statements, such as the one in Figure 5.4, demonstrate an opportunity enable querying over different dimensions of the data. Changes for specific columns can be queries as easily as individual rows.

The ability to link changes of multiple versions together results as a side effect of the model construction. Continuously linked changes opens up avenues of exploration to follow change as it propagates through versions. While change logs will provide a more focused comparison, a triple store with a multi-version graph would give a view of the work through time. Considering the Noble Gas data set's versioning graph's size, many versions may be difficult to store with large, volatile data sets.

The MBVL data set demonstrates a case where versioning graphs can be used to compare the performance of different taxonomy/algorithm pairings. The ability derives from sub-classing each of the AIM changes to give a better perspective where the pairings differed. This approach of extending the versioning graphs adds domain knowledge to the version comparison and helps contextualize the observed differences.

CHAPTER 6
DETERMINING DATA VOLATILITY

6.1 Introduction

Change distances opens new avenues to study and communicate data set change beyond the surface analysis provided by version labeling. The bar graph in Figure 5.5 showing the amount of change in each GCMD version release only shows the relation of the changes to each version and disconnects the changes from time. The bar graph presents a flawed view because the evenly spaced bars imply the distance, in time, between version releases are the same. Such a view of how the versions are released suggests that not only is the change distance between versions large, but that the change rate is also large. The change rate communicates expectations for the data consumer on the value of a data set since committing to a data set that will soon be invalidated is problematic. The following chapter addresses data volatility, the likelihood of data change, and how traditional versions can hide the actual change rate of data sets.

6.2 Determining Volatility

Instead of charting the version changes in evenly wide bars, the versions are spread across time based on the time of publication to the KMS as seen in Figure 6.1. Since each of the versions were dominated by the addition counts, the count is divided by the number of days between the publication of a version on the left side of the line and the release of the replacement version on the right side of the line. The height of the line on the chart gives the amortized rate of change until the release of the new version. The area underneath the line is the total amount of change the new version introduces. Since each version packages together all the changes into a single release, the actual change rate is unknown.

Three observable clusters appear in the time aware presentation of the versions, highlighted in Figure 6.2. According to the *Keyword Governance and Community Guide Document* [101], "Full GCMD keywords list releases get a new major version

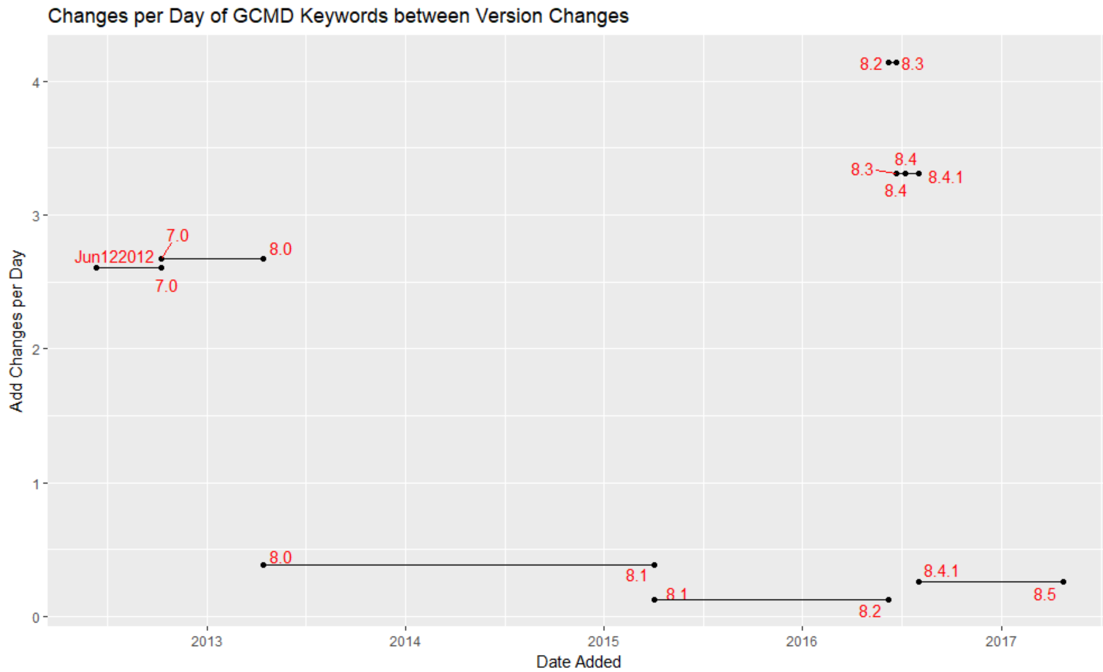

Figure 6.1: Add counts for all versions of GCMD up to 8.5 evenly distributed over the time of version validity.

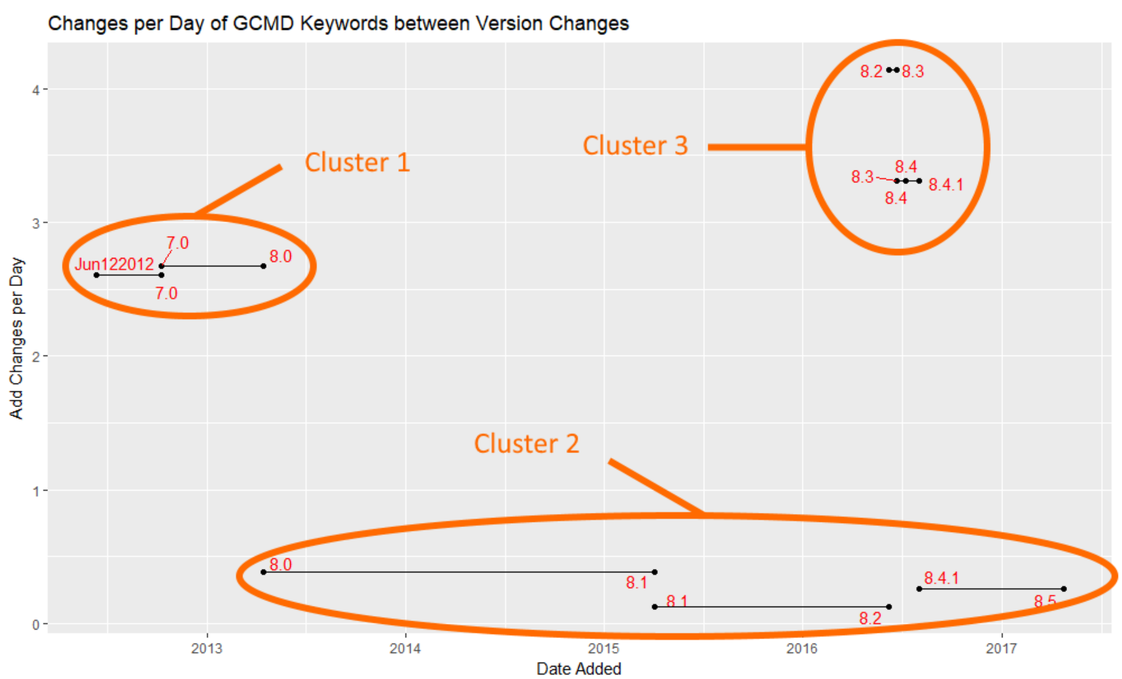

Figure 6.2: The change rate of different Global Change Master Directory versions organized into three visible clusters. Cluster 2 denotes a sudden burst of version releases which is notable.

Table 6.1: Global Change Master Directory versions with old start time changes.

Version Name	Publish Date	2008	2014	2015
8.2	June 7, 2016	0	4	2
8.3	June 21, 2016	0	7	1
8.4	July 7, 2016	5	0	1

number (e.g., 8.0). Incremental releases for updates to topics, terms, and variables get a new minor version number (e.g., 8.1). The statement explains the activity in Cluster 1 where there are sufficient changes to warrant a full release of the keywords. Cluster 2 captures the change rate and duration of minor versions, except those from 8.2 to 8.4.1 which are in Cluster 3. Cluster 3 demonstrates a flurry of activity occurring between June 7, 2016, to August 2, 2016. Considering the previous pattern of taking at least six months between releases, three minor version releases within as many months is highly unusual.

An immediate concern is that Cluster 3 does not result from a sudden burst of activity, necessitating rapid version replacement. An email inquiry (T. Stevens, personal communication, May 2, 2018) into reasoning behind the successive publication returned a statement that "Our government customer had directed us to release the keywords all at once this way." Another way to dig into the behavior is to look into the impact assessments accompanying the versions. Impact assessments prior to Version 8.5 are not publicly available, and only assessments for versions 8.2, 8.3, and 8.4 were received upon request. Of the 6 requests affecting Earth Science Keywords in 8.2, published June 7, 2016, 4 were made in 2014, and the remaining 2 were made in 2015. Version 8.3 had 8 entries in its impact assessment with 7 entries originating in 2014, and the remaining entry from 2015. The 6 entries 8.4s impact assessment has 5 entries from 2008 and 1 entry from 2015. The data is collected in Table 6.1.

6.3 Earth Observing Laboratory

The Earth Observing Laboratory (EOL) of the National Center for Atmospheric Research (NCAR) distributes small data sets, around 10-12 files per data set, regarding lower atmospheric data beginning in 2005 [102]. The EOL data sets

Table 6.2: Version Aggregate of Earth Observing Laboratory Data Sets

Number of Versions	Number of Data Sets
1	1155
2	141
3	26
4	10
5	3
Total	1335

are unique in the data set size means management often does not require automation. In mid-2014, EOL began assigning version to stored data sets. When receiving a new version of a data set from a researcher, the practice is to upload the entire new data set, and replace all old files.

The meta-data extraction and analysis was done by the script found in Appendix H. The script organizes the information into classes representing the data set, the versions of the data set, and the files within the versions. The nested class structure allows federated data computation by having each data set manage version analysis within the data set object. The versions also become grouped and co-located with associated versions in the same data set.

Of the 1335 data sets maintained by EOL with versions, only 180 data sets had more than one version. The full distribution of version counts is in Table 6.2 The 1155 other data sets were filtered out since change distance could not be computed for single-version collections. Since all the files are replaced on an update and a unique file identifier like a hash sum was unavailable, file matching between versions rely on filenames to perform change mappings. For all files that matched names across versions, the relation was classified as modification. This approach will over-count the number of modifies, but provides an upper bound on the data set volatility in the repository. Each count is then normalized by the number of files in the previous version to standardize comparison between data sets regardless of data set size. The average for each data set is taken for each AIM change.

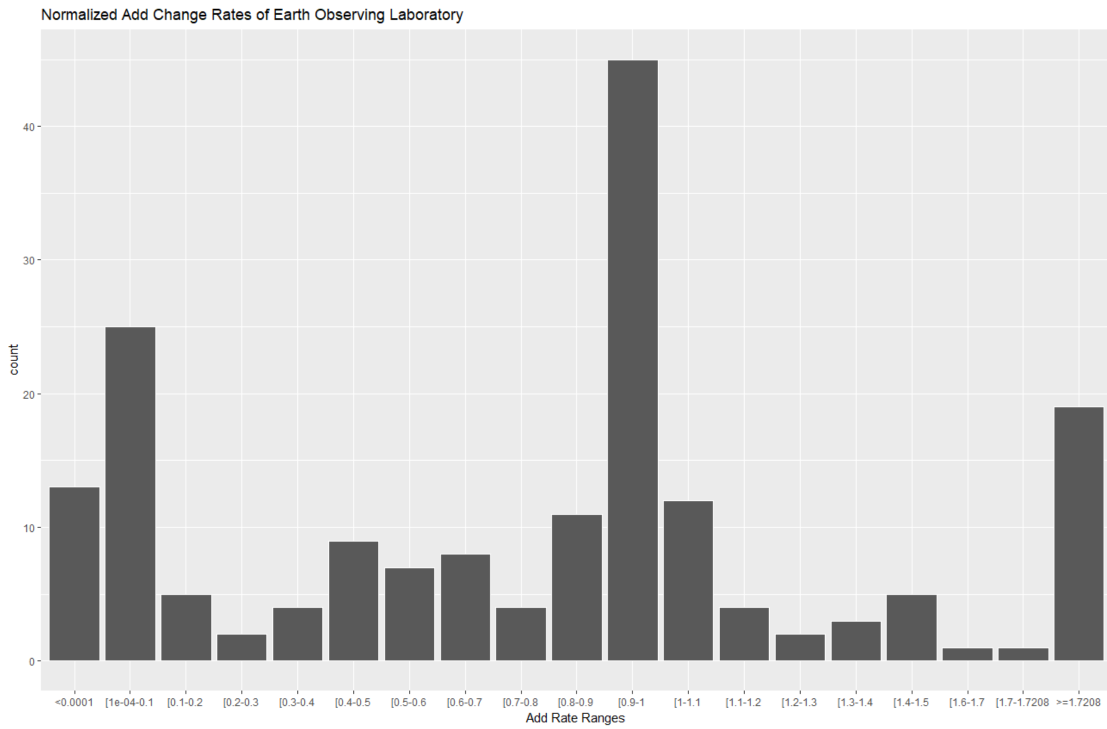

Figure 6.3: Distribution of average normalized Add counts for each data set in Eath Observing Laboratory.

6.3.1 EOL Versioning Behavior

Given that EOL replaces the entire old data set when updating, the expected behavior of the transitions would be modifies concentrating close to 1 and additions and invalidations distributed close to 0. The assumption is that researchers have little reason to change the file naming scheme. The data surprisingly indicates that data sets in EOL primarily gravitate towards addition and invalidation values of 1. modification counts score more close to 0 in a complete reversal of expectations.

Figure 6.3 shows the distribution of addition scores. The primary feature of the chart is the bar situated in the '[0.9-1' range, meaning that about 45 data sets add a number of files equal to the original size of the data set. Secondary features include the bars on the far right and far left of the chart, but the bar on the right side is a collection of outliers. In the outlier data sets, the size of the data set increased drastically compared to the behavior of other data sets managed by EOL. Outliers are determined by collecting values above 1.5 times the InterQuartile Range (IQR) showing in Table 6.3. A more muted distribution appears around the 0.5 mark

Table 6.3: Normalized Change Statistics

Stat	Add	Invalidate	Modify
Mean	0.714312707	0.654819294	0.345180706
Std. Dev	0.509878564	0.420093557	0.420093557
Min	0	0	0
Q1	0.28635075	0.142857	0
Med	0.9146635	0.9642855	0.0357145
Q3	1.00358625	1	0.857143
Max	54.25	1	1
IQR	0.7172355	0.857143	0.857143

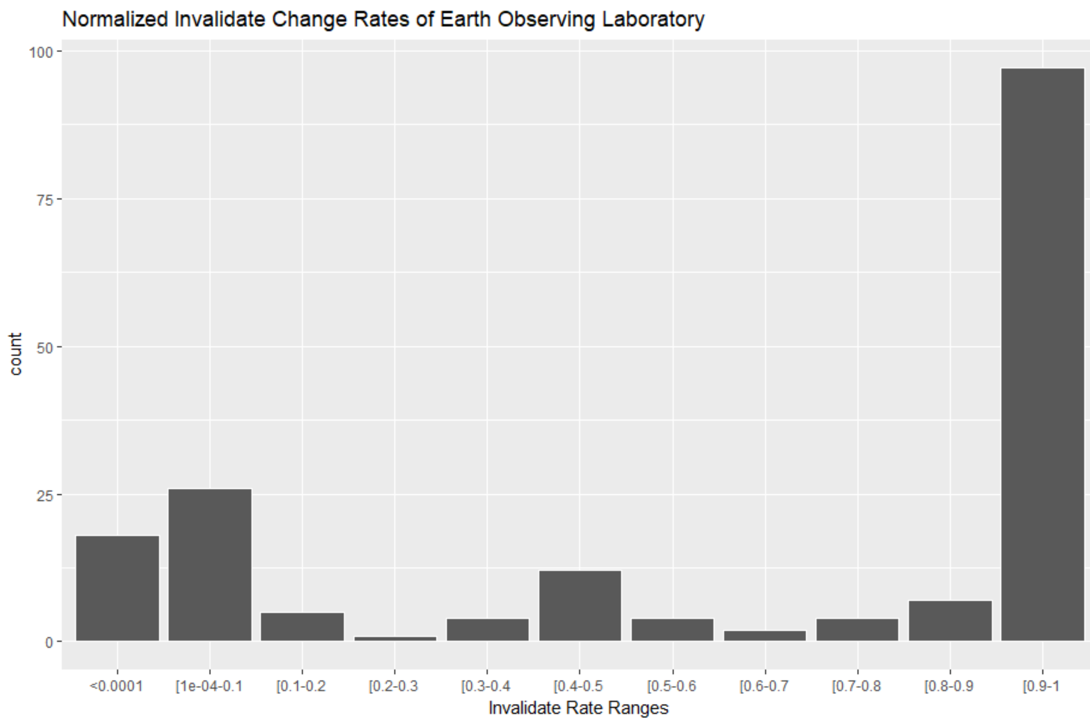

Figure 6.4: Distribution of average normalized Invalidate counts for each data set in Eath Observing Laboratory.

where data sets grow more gradually.

The normalized invalidation score in Figure 6.4 shows a majority of data sets removing all or almost all files in the data set. Coupled with the information that a quarter of the data sets added close to the original data sets' size of files suggests that the entire data set is being replaced. Invalidations do not have outliers since only files within the data set can be removed. The invalidation distribution is extremely biased with only 0.04 separating the median and maximum value. From Table 6.3,

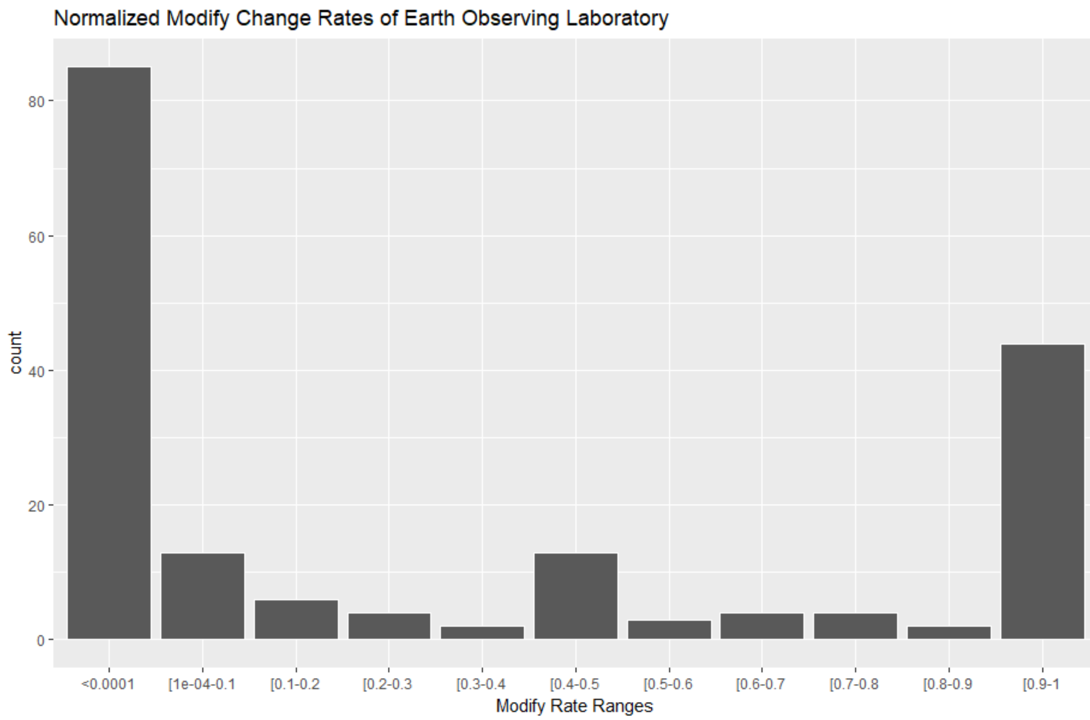

Figure 6.5: Distribution of average normalized Modify counts of each data set in Eath Observing Laboratory.

at least a quarter of values are 1. Figure 6.4 also shows a muted distrubtion around 0.5.

Figure 6.5, representing the normalized modification distribution, is almost a mirror of the invalidation chart. The right bar is specifically cut off to capture only 0s, showing that almost a majority of data sets modify 0 files, having 0 files that share names between versions. The distribution is consistent with a practice of removing all the files in a data set and replacing the files with a new data set using different filenames. The second feature of this graph shows around 40 data sets in which all or almost all files match across versions. A small spike of data sets are centralized around 0.5, very much like the other normalized change graphs.

The high concentration of data sets towards 1 in additions and invalidations suggests a more complicated interaction within the data sets. Individually, the normalized distributions do not show the connection between all three changes since the changes share a common feature, the version transition the changes describe. Together, the AIM changes create a coordinate in three dimensional space, showing

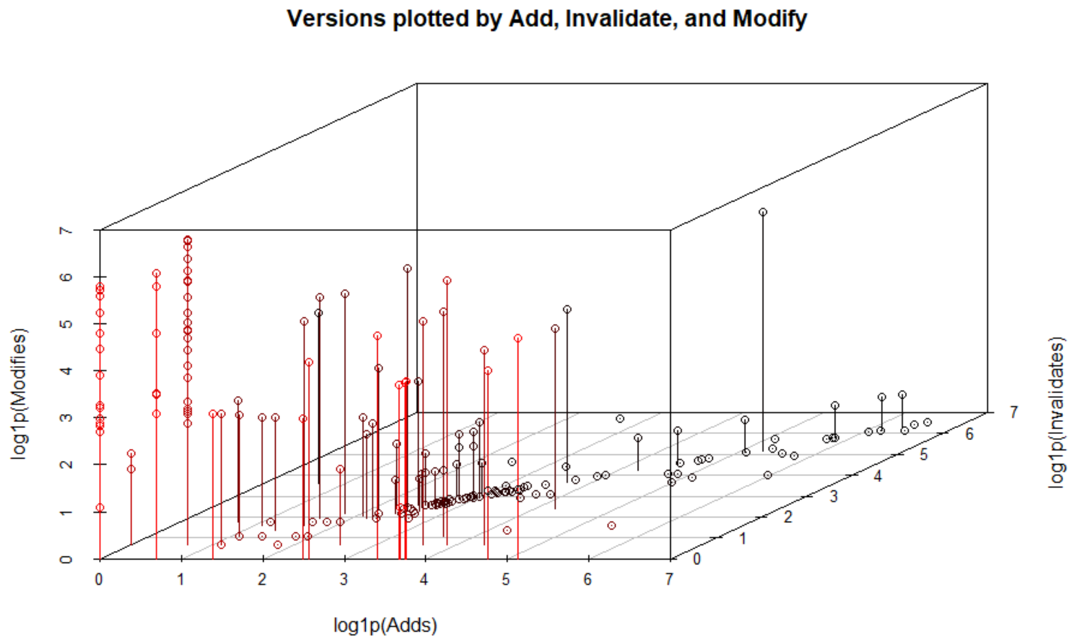

Figure 6.6: Versions in the Earth Observing Laboratory data collection plotted as a combination of **Add**, **Invalidate**, and **Modify** counts.

the inter-relation of the changes. Figure 6.6 shows a scatter plot grouping unnormalized change distances for each version. Unlike the other charts, the size of the changes are not normalized by data set size, but the values have the log1p function applied to account for a heavy bias towards 12 and 13. Notice the one-to-one trend between additions and invalidations which shows the tendency of data sets to replace every file and assign a new filename. If the two changes did not co-occur, a normalized addition score of 1 would indicate that data sets tend to double in size instead. The files are more likely to retain filenames when only a few files in a data set are being modified.

6.4 Analysis

6.4.1 Impact Assessment Change Counts

The impact assessments obtained for GCMD Version 8.2, 8.3, and 8.4 revealed that some of the changes began much earlier than the valid duration of the previous version. Adjusting for the new duration, the change rates reduce to under 0.5, align-

Table 6.4: Differences in VersOn and Impact Assessment metrics

Version	Add	Invalidate	Modify
8.2(VO)	53	1	26
-8.2(IA)	48	0	4
	5	**1**	**22**
8.3(VO)	58	0	13
-8.3(IA)	58	0	10
	0	**0**	**3**
8.4(VO)	53	0	1
-8.4(IA)	66	0	5
	-13	**0**	**-4**
8.5(VO)	68	2	22
-8.5(IA)	55	0	30
	13	**2**	**-8**

ing with the values in Cluster 2 in Figure 6.2. The impact assessments furthermore provide change counts in the format of addition, invalidation, and modification. In none of the cases, shown in Table 6.4, did the metrics completely align although Version 8.3 came close with a difference of 3. VersOn does not consistently overestimate or underestimate across the versions, but the assessments most consistently align on invalidation which make up very few of the changes.

An investigation into the specific differences in Version 8.3 revealed that the term “Saline Lake” does not appear in the change log, but “Leaf Area Index (LAI)” appears twice. LAI appears twice because it has two unique identifiers. Six terms appeared in the change log as modification but missed three terms from the impact assessment. The primary driver between the differences lies in impact assessments being sourced from community requests. The focus of the change analysis becomes arranged around the preferred label rather than the unique identifier used to implement the keyword. Impact assessments capture changes that modify a keyword’s label and that doesn’t change the keyword’s place in the taxonomy as a result. The change log uses a keyword’s unique identifier and its placement in the taxonomy to determine changes to the structure. The difference in metrics collection once again illustrates the producer/consumer dynamic in data version management as well as the need for clear versioning practices. While the comparison between the two counts would be invalid due to differences in practice, a valid comparison could

Table 6.5: Summary of Kolmogorov-Smirnov Test results for Earth Observing Laboratory.

	Add	Invalidate	Modify	Versions
Length	205	192	114	227
D-Value	0.12919	0.14464	0.19727	NA
p-Value	0.05487	0.02575	0.005443	NA

be constructed using the unique identifier to align entries and just the preferred label to determine if the keywords differ.

6.4.2 Hidden Volatility

To determine if the actual change rate is being obscured by the version publication rate, the duration of each version must be calculated. Since only one version is published at the end of that duration, simply taking the inverse of the duration will give the version publication rate. The change counts are then multiplied by the version publication rate for each version to acquire the change rate for each version. Because the change distances are often the size of the data sets, as shown in Figure 6.6, the means must be adjusted to align with the version publication rate mean in order to perform the Kolmogorov-Smirnov test. The test will determine the likelihood that the change distribution comes from the same distribution that produced the version publication rates.

Each version of a data set stored in EOL is assigned three different times, "version publish time," "version creation time," and "version modification time." Version publish time indicates the time the version was made available to the public, usually the data set was added to the database. Version creation time denotes the moment at which a version designation was given to the collection of files, beginning in mid-2014 when the versioning system was implemented. Version modification time indicates the time at which the version metadata was changed. Using version publish time most closely resembles the duration of version validity, and the following computations use version publish time.

The duration is calculated by taking the publish time of the next version and subtracting the publish time of the current version. Some of the durations needed to be filtered out to provide valid results. Due to a few coding errors in time

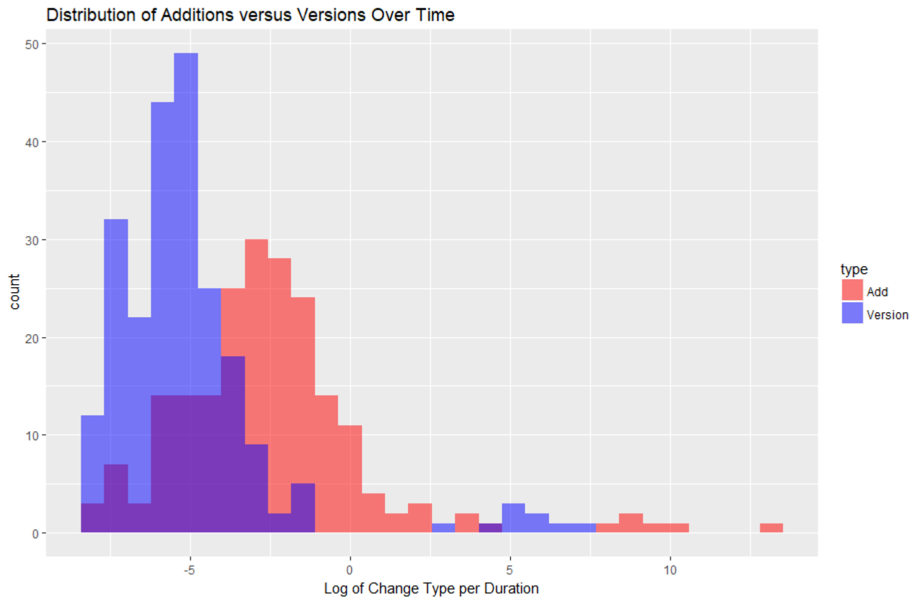

Figure 6.7: Distribution of **Add** rates of each version against the version publication rate in Earth Observing Laboratory.

assignments, 7 versions had to be removed because the durations were negative. Duration is measured in days, and the rate of version publication is determined by taking the inverse of the duration, giving versions per day. To acquire the AIM change rates, the changes are divided by the associated duration for each version, returning change per day. Since the change rates are biased towards 0, the log of the rates are taken to give the values a more normal distribution. Values where an AIM change is 0 had to be removed in order to properly apply the log function. The size of each distribution can be found in Table 6.5.

Since the durations are log-normally distributed, concentrated close to 0, the log of the durations are taken to normalize the data. The log function is also applied to the AIM changes after the division by the duration. The inverse of the log of the duration is taken to acquire the rate of version release. The addition distribution in Figure 6.7 and the invalidation distribution in Figure 6.8 are translated slightly a few points to the right of the darker version distribution. As noted in Section 6.3.1, almost half of the versions have 0 modifies, meaning that the values must be filtered out. The resulting graph for comparison in Figure 6.9 shows a more flattened modification distribution.

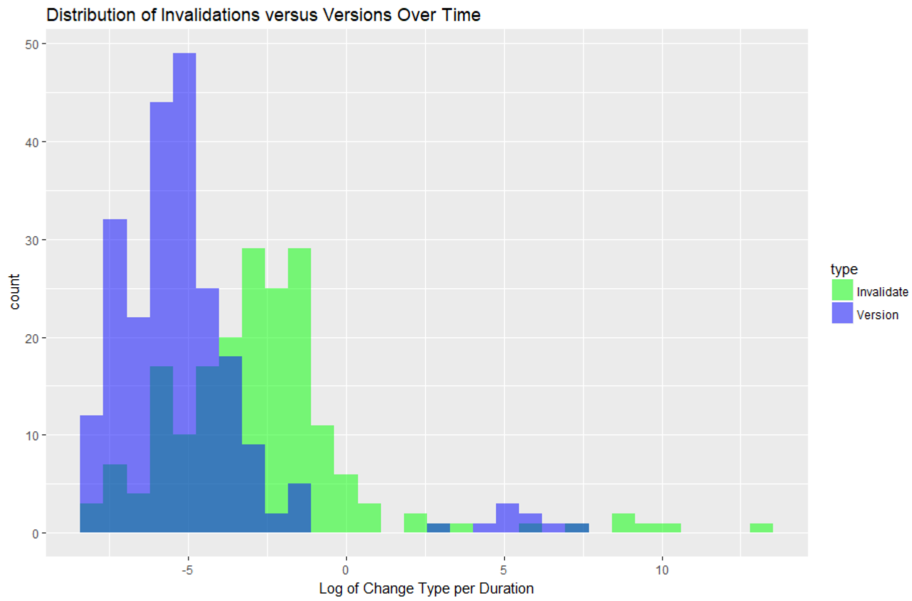

Figure 6.8: Distribution of **Invalidate** rate of each version against the version publication rate set in Eath Observing Laboratory.

The Kolomogorov-Smirnov Test was used to determine if the additions, invalidations, or modifies follow a distribution the same as the version publication distribution as the null hypothesis. The change distributions were translated down such that the version mean and the change mean aligned in order for the test to produce meaningful results. Table 6.5 shows that the distribution of additions is statistically significant with 90% confidence while invalidations and modifies can reject the null hypothesis with at least 95% confidence. The analysis demonstrates that there is strong evidence version publications do not accurately reflect the actual change rate of the EOL data sets.

Overlaying the change distributions in Figure 6.10, notice the distributions are all log-normally distributed around the same mean. The normality indicates that data users can reliably predict the rate of addition or invalidation for a data set, setting an expectation for the valid duration of a version the data consumer currently uses. The distribution additionally allows a data consumer to predict an expected amount of change in a data set after not updating the data for a period of time. The flatter modification density curve shows a bi-modal behavior, suggesting that modification changes follow two separate distributions. Remember that the modification counts reflect an upper-bound value of change in versions and many data sets completely replace the files in a version. The two distributions likely reflect the behavior of versions of differing maturity with older versions of data sets requiring fewer modifications to the data, resulting in a lower change rate. Drawing expectations from Figure 6.10 should be done carefully since Section 6.3.1 shows that modification does not correlate the same way that addition and invalidation do. The different behavior means that the change rates are not independent distributions.

6.5 Summary

Implementing the versioning model, VersOn, yielded results more complicated than the simple model expected. While VersOn addresses difficulties in other linked data approaches, it requires many more triples to express the relationship. The scalability created space issues with encoded change logs, especially in JSON-LD. RDFa also proved to be a more restrictive structured data method than expected. The implementation required multiple attributes per modification to accommodate

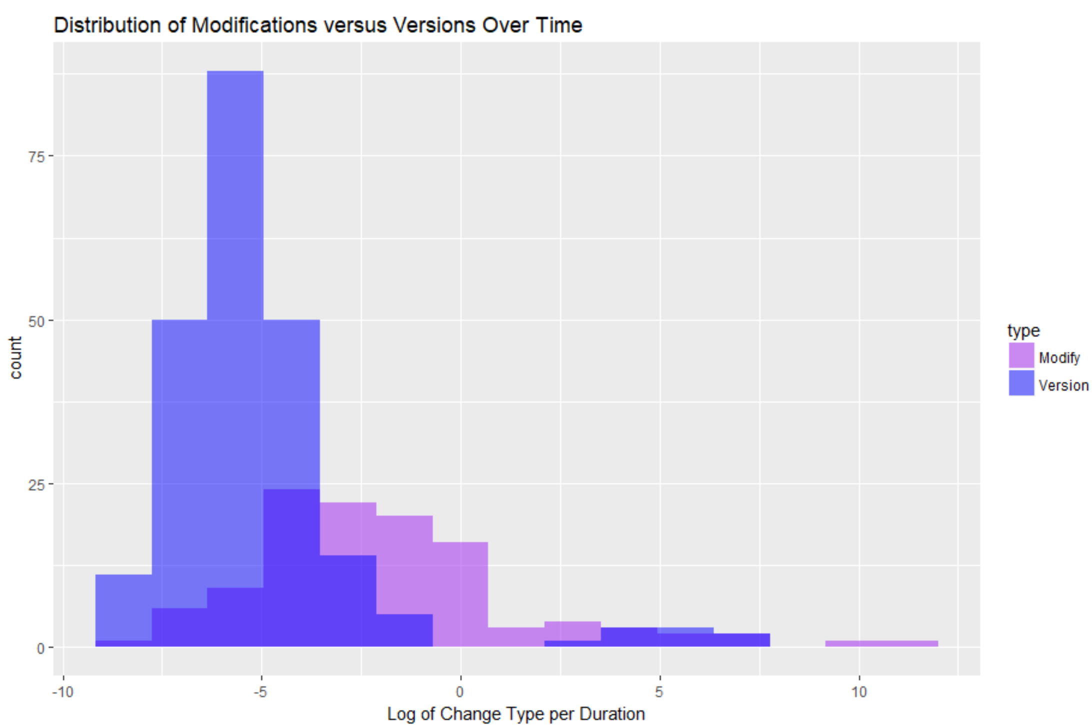

Figure 6.9: Distribution of average normalized Modify counts of each data set in Eath Observing Laboratory.

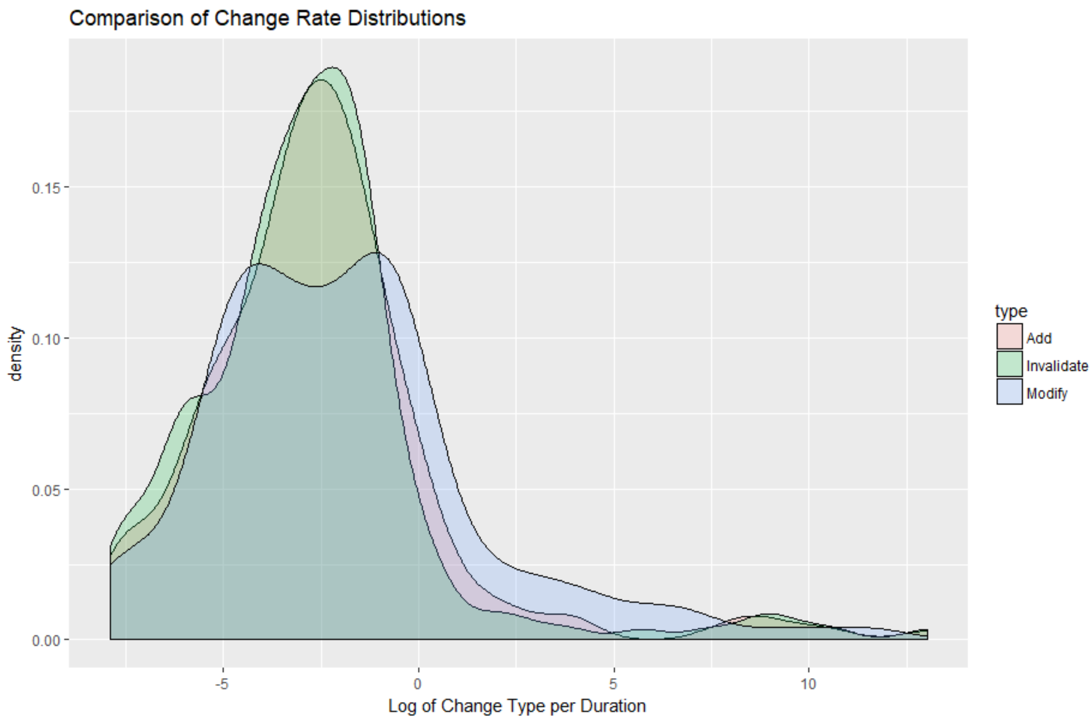

Figure 6.10: Density plot of change rates in Eath Observing Laboratory divided by type.

both row and column attribute associating with a cell. There were discrepancies between GCMD Keyword version identifiers and the change detected within the data set. Finally, the versioning model was used not to document sequential versions, but to compare the results of different species classifiers.

6.5.1 Model

The versioning model's development began with an expectation that versions would be sequential. The MBVL data set demonstrated a case where four data sets were not related by temporal sequence. One is not a transformation of another since we are studying the effects of changing the taxonomy or algorithm. Additionally, since we do not know which version is the best, we cannot consider any data set as an update of the others. Finally, no entity preexisted as the data sets resulted from an ongoing analysis and further steps have not been developed. As a result, the current definition of *prov:wasDerivedFrom* would not be able to capture the relationship between these data sets. VersOn improves upon expressing versions in linked-data by focusing on the differences between objects rather than the sequence. VersOn takes inspiration from *schema:UpdateAction* by dividing up the changes into three forms, but improves upon it by adopting the provenance model's transition from one object to the next. The resulting forms diverge from Schema.org's context of an agent acting upon an object.

The reason *prov:Generation* and *prov:Invalidation* are not used is because they expect an activity to act upon an object. It is not generally true that an action actively adds or removes an object's attribute from in the left-hand version to produce the right-hand version. That assumption minimizes the ability to conduct versioning comparisons between objects that are not sequentially adjacent. The PROV-O concepts also have a property pointing towards the responsible activity which is assumed to be the immediately preceding activity. The assumption fails to consider the case where a change propagates further changes downstream, generating or invalidating the current object. VersOn avoids confusion by only considering the versions and their differences.

6.5.2 Implementation

6.5.2.1 Scalability

VersOn breaks up a revision into constituent changes, acting upon different attributes of the version. Other ontologies use a single property to relate versions. While it is more specific, the VersOn implementation encounters scalable space consumption problems. PROV-O only requires 3 to 5 triples in order to make a *prov:wasRevisionOf* statement. This model uses 9 triples for a *vo:ModifyChange* and 7 to encode *vo:AddChange* and *vo:InvalidateChange*. An implementation of the model, therefore, has space complexity of

$$O(7M + 5(A + I))$$

since declaring version objects takes a constant two statements. However, a similar structure can be achieved using *prov:wasDerivedFrom* to replace modifications and *schema:AddAction* and *schema:DeleteAction* to replace additions and invalidations.

The resulting space complexity is:

$$O(7M + 3A + 5I)$$

This is fairly similar with additions seeing a reduction since the left-hand version no longer contributes to the *schema:AddAction*. Thus the primary benefit of using this model comes from semantics.

6.5.2.2 Structured Data and the Model

While machine-readable change logs have always been a desired goal of this dissertation work, their requirements diverged from the versioning model's needs. The model, as a result, leverages very little from visible content on the change log. Symmetric representation in the change log also made encoding the versioning graph using RDFa challenging without explicitly defining the whole versioning graph in invisible span tags. Adherence with a change log oriented approach would also likely have reduced the number of statements needed to form the versioning graph. The resulting ontology would likely be a collection of properties and concepts to use in annotating a document.

The VersOn construction provides great flexibility for version and change distance capture. VersOn adapts to multiple attributes smoothly. Greater adherence to structured data adoption may need to come in the form of graph simplification or metered release of new editions to ensure that change logs do not grow too large.

6.5.3 Distance Measure

As mentioned in Section 1.4, a version model provides the framework, provenance models provide the context, and change logs fill in the gap between versions. change logs, therefore, provide the most substance to quantify the distance between version. The automated change log generation additionally ensures this by including all differences into the change log. Anything unmapped remains the same between versions and does not contribute towards the distance. While MBVL demonstrated a case where domain knowledge could be added to the versioning graph and provide context for distances, other applications may not demonstrate the same amount of uniformity within changes. More domain information and reasoning may be necessary to determine if one addition significantly more impactful than others in a versioning graph.

6.5.4 Volatility

In Chapter 6, we explored different ways in which versions hide the actual change behavior within a data system. The GCMD keywords showed one perspective when evenly distributed, but spread across time, the versions have very different behavior. The change rates, once re-coupled to time, show that versions can provide a misleading view of how changes apply to a data set. versions package together changes that can originate from times prior to the previous version, disrupting the assumed relationship between consecutive versions. GCMD also breaks assumptions when the impact assessments use different metrics to determine a data set change, reinforcing the concept that perspective and context play a major role in versioning methodology. Using only version names, EOL data is distributed uniformly by single versions, but 6.6 shows a more vibrant behavior in the data sets. The chart revealed trends in file naming and replacement. The data sets also demonstrated that AIM changes behave significantly differently than the behavior versions reveal.

CHAPTER 7
DISCUSSION & CONCLUSION

7.1 Introduction

Based on the results presented in Chapter 3, the versioning model more completely captures the changes occurring between versions by preserving version continuity, capturing Addition, Invalidation, and Modification (AIM) changes, and directly connection attributes with versions. The results in Chapter 4 showed that the impact of encoding VersOn into change logs exceeds the expected 50% decrease in performance. VersOn produced results in Chapter 5 indicating an increase in change distance precision using the ontology by enumerating the changes between dot-decimal identifiers. The results from Chapter 6 highlights the increased accuracy in capturing data set change rates after factoring time into the change distances.

7.2 Versioning Graph Completeness

Three requirements were identified to obtain completeness in a versioning graph:

1. Captures addition, invalidation, and modification (AIM) change modeling
2. Links attributes to versions
3. Maintains version continuity

A provenance based model, discussed in Chapter 3, makes the new version responsible for all differences in a version change, partially satisfied the second and third requirements. Old attributes, as seen in Figure 3.1, invalidated by the new version were incorrectly linked to the appropriate version, but the changing attribute shared between Version 1 and Version 2 is correctly linked. For example, entry Tur030 in Figure 5.4 would be associated with Version 3 of the Noble Gas database even though the attribute no longer appears in the version. The changing attribute, however, does not provide correct version continuity when a third version is added.

All versions become associated with a singular attribute which makes the proper ordering of the pre- and post-values ambiguous.

A log based model, shown in Figure 3.2, satisfied the change modeling requirement, but fails to meet the second and third requirements. Because all the changes are associated with the log, attributes are not linked to the original data versions, and there is no way to preserve version continuity through the model. Placing the log model between versions in a hybrid approach, seen in Figure 3.3, re-establishes continuity between versions, where a third version can be added without violating or confusing the concepts in the previous versions. The attribute to version connection remains absent in the hybrid construction since attributes are still associated with the Log concept instead of the versions.

A fully connected model satisfied the attribute to version connectivity requirement and the version continuity requirement. The attributes were directly associated with the appropriate version and continuity through versions was achieved through the relation between attributes. As changes link attributes across versions, the attributes imply a continuity because each attribute must be a part of a version. The relationship is the same one seen in the **modification** change in Figure 5.3 and Figure 5.4. The fully connected model does not satisfy the requirement to distinguish between the AIM changes. The formulation has a single change, but cannot model missing attributes due to addition and invalidation.

The final model, implemented as VersOn, separates the fully connected model into three formulations to satisfy all three requirements. The addition and invalidation formulations remove an attribute to successfully model the associated AIM changes, and uses the version-change relation to maintain version continuity. The modification formulation maintains continuity through versions by the implied relation across attributes in the model. Attributes in each of the forms are directly associated with the appropriate versions to satisfy the second requirement.

7.3 Effects on Change Log Performance

Prior work from Buneman [13] and the close relation to provenance data suggested that encoding linked-data change information into a change log would decrease performance of the log by more than 50%. The findings in Section 4.3 show that change data does not consistently utilize space on the order of the original data set. In Table 4.1, the un-encoded change log is a little over double the size of the original data file. The RDFa change log was then more than ten times the size of the text-only log, meaning that the performance of the change log has decreased by at least 90%. The JSON-LD change log was even larger, over twenty times the size of the text-only log, resulting in a 95% decrease to performance. In both of the encoded logs, the impact to performance is significantly greater than the expect 50%.

The impact to performance continues to be seen in Table 4.2 and Table 4.4 where the text-only change logs are smaller than the original data file. The RDFa change logs have consistently 90% reductions to performance and the JSON-LD change logs reduce performance by approximately 95%. Looking at Table 4.3 and Table 4.6, notice that the version changes are dominated by **modification** changes. A comparison in the Copper Minerals Database show that **addition** and **invalidation** changes gain large compression benefits from row and column summarization, but **modification** cannot summarize the changes it captures. Combining the dominance and compression observations, **modifications** play a significant role in failure to meet change log performance expectations. The performance reduction does not entirely restrict adoption since a single script, as seen in Appendix C, can dynamically generate change logs for a number of versions due to standardization. The responsibility of log storage does not entirely lie with the data producer, but may be adopted by the data consumer as needed or dynamically generated.

A finding through the encoding process is that RDFa introduces a number of challenges to creating a linked-data change log compared to JSON-LD. The RDFa functions by annotating visible data in the document, but VersOn utilizes very little of the visible content, meaning RDFa forces some change content to be adjusted to conform with the XML or HTML structure. JSON-LD frees the VersOn from the strict structure of the visible change log document, allowing the intended expression of the model. The trade-off to achieve more accurate version model content results in increased storage space.

7.4 Increases to Change Distance Precision

The GCMD Keyword Governance and Community Guide Document defines the requirements for labeling major and minor releases using dot-decimal identifiers [101]. Figure 5.5 shows that VersOn is able to provide more precise counts, showing that major version changes differed by three hundred to five hundred **additions**. VersOn, additionally, characterizes GCMD Keywords as an addition dominated data set. In each of the minor version changes between GCMD Keyword versions, VersOn is able to more precisely distinguish between the number of changes separating the versions than a single increment of the minor version number. In the MBVL data set, VersOn is able to support the comparison within the AIM changes, enabling increased precision without any modifications to the base ontology. The additional precision using taxonomic ranks was achieved through sub-classing the AIM change concepts in the ontology. Increased precision in change distance from VersOn enables more detailed comparisons between versions rather than the categorical comparisons currently enabled by dot-decimal identifiers. VersOn change counts enables the computation of other attributes more precisely such as a data set's rate of change.

7.5 Increases to Change Rate Accuracy

After redistributing GCMD Keyword changes across time, the change rates organized into three clusters as seen in Figure 6.2. In the third cluster, versions were released at a rapid pace, less than a month apart. The changes implemented by the Versions 8.2 to 8.4 were found to originate from years prior to the release of the immediately previous version. The changes are not restricted to the valid time frame of the immediately previous version. Using the starting time of the change rather than the preceeding version, the change rates of Cluster 3 match the rates in Cluster 2. Using the change time instead of the version publication time to compute the VersOn change rate increases the accuracy of the change rate measurements.

The EOL collects a large number of small-sized data sets, around thirteen hundred, but just over a thousand of the data sets have only one version, making the data sets un-versionable by VersOn. Of the remaining 180 data sets, distribution of version publication rates was compared to each of the AIM change rates using the Kolmogorov-Smirnov test. The test compares the cumulative distribution functions of each distribution to find the largest different to determine likelihood both functions come from the same distribution. The version publication rate does not match the distribution of any of the change rates, and because VersOn captures the changes between versions, VersOn change rates represented the more accurate change rate. VersOn allows data consumers to better understand the actual data set rate of change, enabling connected systems to form better expectations of changing data's effects.

7.6 Producer/Consumer Versioning Dynamic

In the diagrams in Figure 1.3 and Figure 1.4, notice that all the information going into the Versioning System originates from the Producer. Even in the one interaction originating from the Consumer, the Consumer does not input any information into the system. In the use case, the Producer poses sole authority over defining the change information. Looking at dot-decimal identifiers, the Producer once again posses sole authority on determining the interval to increment. Because the identifier is also a data set label, a Producer must finalize the identifier prior to publication, meaning the Consumer must use the label regardless of the amount of pertinent change.

In the GCMD Keywords version change from 8.4.1 to 8.5 in Figure 5.4.2, we saw that the amount of change varies greatly depending on the Consumer utilization of the data set. In the Bridged method, the version change is modification dominant, a vastly different behavior than all prior version changes. The Standard method used URIs to match attributes, but the method resulted in a total replacement of the data set. We know that GCMD Keywords used a third method of assessing the change because the Standard and Bridged methods constitute a full keyword release, requiring a new major version label. VersOn allows data Consumers to self-assess the amount of change created by a version change rather than being forced to use the assessment of the producer at the time of publication. VersOn also increases the transparency of the methods that the Producer used to make a conclusion of the change distance, resulting in a particular version identifier.

The need for versioning method transparency becomes especially apparent in Table 6.4. The impact assessments and VersOn both use addition, invalidation, and modifications to count the number of changes made in the resulting version. The assessments, however, are based on requests made to the keywords group while VersOn used URIs and taxonomy structure to determine counts. The difference in methods result in conflicting assessments to the amount of change and effected keywords. The implementation of VersOn clarifies the distinction by making the mapping method between versions to assess change explicit and visible.

In the Earth Observing Laboratory, the versioning method is not apparent the Consumer. Each data set has very few versions with minor version name differences, but a majority of the data sets are entirely replace or at least use entirely different filenames as seen in the linear trend between **additions** and **invalidations** in Figure 6.6. The pattern of behavior between versions is hidden or inconsistent between data sets collected by the laboratory.

7.7 New Versioning Nomenclature

Analysis of versioned data sets has revealed three types of data, dependent on the way in which versions are released: single, periodic, and intermittent. Single version data sets contain data which cannot be replicated or in which modification would entirely invalidate the data. High energy physics, previously mentioned, and surveillance data fit within this category. The data sets in this category will usually only experience additions and invalidations since scientists cannot change the data.

Periodic data sets exhibit version releases at regular intervals in time. Large data collections usually exhibit a regular behavior when they follow a periodic data collection scheme. The ARM data center releases data at daily intervals, meaning new versions every day. The reasons that ARM data sets are not overloaded with version numbers is that some operations, in this case new files, are masked to increase the pertinence of each version designation. The problem that masking additions causes is the actual amount of change within the data set over time also becomes masked. The data set then appears to be intermittent when it actually undergoes periodic changes. As seen in GCMD Keywords and EOL, changes are not necessarily evenly distributed among versions. The changes, as a result, are also not evenly distributed across time. As mentioned with distributed versioning methods, periodic version releases can be used to control the volatility of a data set by collecting many changes over time before publication. Periodic data sets expend version identifiers very quickly since they must release a version even if few significant changes have occurred. Periodic data sets have the benefit of being accurate within a bounded time frame.

The final type of data set follows intermittent versioning which is characterized by releasing versions as appropriate or as necessary. The data sets are not bound by an established release schedules. In the intermittent category falls GCMD Keywords, the Copper data set, and the Noble Gas data set. Irregular version releases allows data managers the freedom to reduce the number of versions necessary to manage the data set. When data managers wait too long to release a new version, the number of changes in a single transition can overwhelm methods to track modifications to the data as seen in the Noble Gas data set. Since intermittent versions are not released based on time, it is very important that versions are released based on some other quantitative measure of change. Failing to do so invites unclear or worse arbitrary distinction between versions. GCMD Keywords define clear requirements for major and minor version releases, but the governance document does not explain the requirements for sub-minor versions which occasionally appear in the keyword repository.

Each data set type can additionally be sub-divided into two categories based on the observations made with the AIM model: Add dominant and Modify dominant. In the data sets currently studied, none exhibit behavior suggesting an Invalidate dominant data set. A data set is either Add or Modify dominant when a majority of versions have a majority of either Adds or Modifies. Add dominance indicates that the data is primarily growing while Modify dominance shows that a data sets coverage is primarily stable but occasionally undergoes adjustments. The GCMD Keywords is an example of an Add dominant data set since all its version transitions are comprised of new concepts. The Noble Gas data set shows modify dominance.

7.8 Conclusion

The work in this thesis has demonstrated that VersOn has been successfully applied to a very diverse set of data sources, ranging from static tabular databases to taxonomic trees. The model we developed, instantiated in VersOn, improved change capture of linked-data models by satisfying the three requirements of completeness. Data change logs were made available as linked-data as a result of the improved completeness. Our process to encode VersOn into change logs was not able to bound change log performance reduction to at most 50% because **modifications** were not summarized over rows or columns. The results showed that the ability to summarize change plays a key role in managing the effects of version model encoding on change log performance. We were able, through VersOn enabled change counts, to increase the precision of change measurements compared to dot-decimal identifiers by enumerating the changes between each identifier. The work with MBVL demonstrated encoding domain knowledge around VersOn enabled more precise classification of changes which are pertinent to the domain. Increased precision led to standardized comparison between versions and the more precise computation of other change attributes like the change rate. Our incorporation of time into the VersOn change counts demonstrated an increased accuracy of data set change rate assessment by disconnecting changes from the distribution across versions. Accurately understanding the data set change rate is important to evaluating the needs of versioning systems to manage changes in a data set. Our work in this thesis additionally found that VersOn improved the transparency of versioning methods used between the data Producer and the data Consumer by revealing the methods used to assess change and label versions. The standardized change log and versioning graph generation using VersOn enabled the data Consumer to assess data set change independent from Producer assessments by providing tools, shown in the Appendices, to compute change counts and generate change logs from the client side of the versioning system. Our analysis of change counts and version publication rates revealed a pattern of classification between different data sets based on the rate of change and the dominant change type which may, in future work, reveal particular patterns or needs in versioning.

CHAPTER 8
FUTURE WORK

VersOn opens a number of different avenues for future versioning research. Approaches to compressing modification statements when capturing change in spreadsheets need to be pursued. The ability to study changes of a collection of data as a vector space was not pursued. Data sets can now be studied by unique characteristics resulting from the development of new versioning versioning nomenclature to describe data sets. Version analysis can be expanded by developing the ideas of change usage compression, vector analysis of change space, and the study of data set characteristics using new versioning nomenclature.

8.1 Modification Compression

Modifications had to include the same number of triples as cells modified in the Copper Minerals database. Fewer triples can be used if changes to entire rows or columns can be summarized in a single statement. Encoded change logs would see significant reductions in storage costs as a result. Summarization by row or column may fail when a few cells do not change so a method to express no change in VersOn must be pursued. Using the smaller number of changes or unmodified values, at most half the length of a row or column in a spreadsheet worth of statements is necessary to describe a change to a row or column, respectively.

8.1.1 Dynamic Change Logs

Users selectively use portions of particularly expansive data sets to filter data down to their region of study. Tools can use the versioning model to identify pertinent sections of a large change log and parse out the extraneous entries. Means to isolate change activities are necessary for users to determine the impact a new version has on the operation of their workflow. The versioning graph can also contribute to the generation of unique change logs to accompany dynamically created data sets. As mentioned in Section 2.5.4, users can dynamically aggregate and filter data sets to produce a new unique set of data, but doing so still requires tracking of differences from the original data set or sets. Further work will need to be done determining requirements to automate change log creation for these data sets.

8.2 Vector analysis of Change Space

The three dimensional scatter plot shown in Figure 6.6 show versions plotted in change space as a combination of AIM counts. All points are an absolute distance and direction from (0,0,0), meaning that each version can be represented as a vector. Multiple versions could be added together to show the resultant change to a data set. Further research is necessary to determine whether vector operations can be used to study version behavior.

8.3 Nested Versioning graphs

In Figure 2.1 and Figure 2.2 use multiple levels to capture versions, but VersOn only uses two. Allowing **attributes** to also be **versions** means that a versioning graph can express variable granularity. A graph captures a modification in a file of a data set managed by EOL. The file would be an **attribute** of **versions** of the EOL data set, but a second graph would capture differences in values within the file, making the file a **version**. The second graph could then be nested in the first by connecting the graphs at the file identifiers.

8.4 Patterns of change behavior based on Change Classes

The change classes of unversionable, irregular, and periodic organize data sets by different practices of versioning. Further work is necessary to determine if there are versioning requirements specific to the type of change class, and determine if the classes exhibit specific patterns of change which could improve change data capture. A particular versioning technique or tradition may not be appropriate for every class of data set.

8.5 Database Context

One area not explored by the work in this dissertation is the context of centralized databases. While they resemble spreadsheets, centralized databases only have a single instance and use multiple tables which are routinely merged to answer queries. The scripts and process used to version spreadsheets would not work on these databases since the data is not instanced. The databases, however, use standardized add, delete, and modify commands which do map to the versioning model. Work remains to be done in studying how these commands can be captured and output as a versioning graph instead of using the script to perform the comparison.

8.6 Summary

Future work remains open to a number of different pursuits. Change logs can be shortened by discovering modifications occurring over an entire column which can be summarized in a single statement. Version analysis can now leverage vector operations to study data set change. Organizing data sets by versioning approach opens new avenues of evaluating the behavior and performance of versioning systems. These approaches were left unexplored by the project's conclusion.

REFERENCES

[1] World Wide Web Consortium. Linked data. World Wide Web Consortium. Accessed: June 26, 2018. [Online]. Available: https://www.w3.org/standards/semanticweb/data

[2] L. Moreau and P. Missier, "PROV-dm: The PROV data model," W3C, W3C Recommendation, Apr. 2013, http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

[3] B. R. Barkstrom, *Data Product Configuration Management and Versioning in Large-Scale Production of Satellite Scientific Data.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 118–133. [Online]. Available: http://dx.doi.org/10.1007/3-540-39195-9_9

[4] T. Lebo, D. McGuinness, and S. Sahoo, "PROV-o: The PROV ontology," W3C, W3C Recommendation, Apr. 2013, http://www.w3.org/TR/2013/REC-prov-o-20130430/.

[5] B. Tagger, "A literature review for the problem of biological data versioning," Online, July 2005. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/btagger/LitReview.pdf

[6] I. S. G. on the Functional Requirements for Bibliographic Records, "Functional requirements for bibliographic records," International Federation of Library Associations and Institutions, Tech. Rep., 2009.

[7] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, *Workflow evolution.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 438–455. [Online]. Available: http://dx.doi.org/10.1007/BFb0019939

[8] U. K. Wiil and D. L. Hicks, "Requirements for development of hypermedia technology for a digital library supporting scholarly work," in *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, ser. SAC '00. New York, NY, USA: ACM, 2000, pp. 607–609. [Online]. Available: http://doi.acm.org/10.1145/338407.338517

[9] R. Cavanaugh, G. Graham, and M. Wilde, "Satisfying the tax collector: Using data provenance as a way to audit data analyses in high energy physics," in *Workshop on Data Lineage and Provenance*, Oct. 2002.

[10] K. S. Baker and L. Yarmey, "Data stewardship: Environmental data curation and a web-of-repositories," *The International Journal of Data Curation*, vol. 4, no. 2, pp. 12–27, 2009.

[11] X. Ma, J. G. Zheng, J. C. Goldstein, S. Zednik, L. Fu, B. Duggan, S. M. Aulenbach, P. West, C. Tilmes, and P. Fox, "Ontology engineering in provenance enablement for the national climate assessment," *Environmental Modelling & Software*, vol. 61, pp. 191 – 205, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364815214002254

[12] A. Capiluppi, P. Lago, and M. Morisio, "Evidences in the evolution of os projects through changelog analyses," in *Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering*, J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, Eds., May 2003, citation: Capiluppi, A., Lago, P., Morisio, M. (2003). ?Evidences in the evolution of OS projects through Changelog Analyses.? in Feller, P., Fitzgerald, B., Hissam, B. Lakhani, K. (eds.) Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering ICSE?03 International Conference on Software Engineering Portland, Oregon May 3-11, 2003. pp.19-24.. [Online]. Available: http://roar.uel.ac.uk/1037/

[13] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 539–550. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142534

[14] B. Barkstrom, *Earth Science Data Management Handbook: Users and User Access*. CRC Press, April 2014, vol. 1. [Online]. Available: https://books.google.com/books?id=pI3rTgEACAAJ

[15] S. Chacon, *Pro Git*, 1st ed. Berkely, CA, USA: Apress, 2009.

[16] M. Branco, D. Cameron, B. Gaidioz, V. Garonne, B. Koblitz, M. Lassnig, R. Rocha, P. Salgado, and T. Wenaus, "Managing atlas data on a petabyte-scale with dq2," *Journal of Physics: Conference Series*, vol. 119, no. 6, p. 062017, 2008. [Online]. Available: http://stacks.iop.org/1742-6596/119/i=6/a=062017

[17] M. S. Mayernik, T. DiLauro, R. Duerr, E. Metsger, A. E. Thessen, and G. S. Choudhury, "Data conservancy provenance, context, and lineage services: Key components for data preservation and curation," *Data Science Journal*, vol. 12, pp. 158–171, 2013.

[18] M. Macduff, B. Lee, and S. Beus, "Versioning complex data," in *2014 IEEE International Congress on Big Data*, June 2014, pp. 788–791.

[19] M. Dummontier, A. J. G. Gray, and M. S. Marshall, "The hcls community profile: Describing datadata, vversion, and distributions," in *Smart Descriptions & Smarter Vocabularies*, 2016. [Online]. Available: https://www.w3.org/2016/11/sdsvoc/SDSVoc16_paper_3

[20] D. German, "Automating the measurement of open source projects," in *In Proceedings of the 3rd Workshop on Open Source Software Engineering*, 2003, pp. 63–67.

[21] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller, "Open-source change logs," *Empirical Softw. Engg.*, vol. 9, no. 3, pp. 197–210, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:EMSE.0000027779.70556.d0

[22] K. Herzig and A. Zeller, "Mining cause-effect-chains from version histories," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, Nov 2011, pp. 60–69.

[23] P. P. da Silva, D. L. McGuinness, and R. Fikes, "A proof markup language for semantic web services," *Information Systems*, vol. 31, no. 45, pp. 381 – 395, 2006, the Semantic Web and Web Services. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437905000281

[24] L. Moreau, J. Freire, J. Futrelle, R. E. McGrath, J. Myers, and P. Paulson, "The open provenance model: An overview," in *International Provenance and Annotation Workshop.* Springer, 2008, pp. 323–326.

[25] Y. Liu, J. Futrelle, J. Myers, A. Rodriguez, and R. Kooper, "A provenance-aware virtual sensor system using the open provenance model," in *2010 International Symposium on Collaborative Technologies and Systems*, May 2010, pp. 330–339.

[26] Y. L. Simmhan, B. Plale, and D. Gannon, "Karma2: Provenance management for data-driven workflows," *Web Services Research for Emerging Applications: Discoveries and Trends: Discoveries and Trends*, p. 317, 2010.

[27] S. Miles and Y. Gil, "PROV model primer," W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-primer-20130430/.

[28] P. Groth and L. Moreau, "PROV-overview," W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

[29] P. Missier, L. Moreau, and J. Cheney, "Constraints of the PROV data model," W3C, W3C Recommendation, Apr. 2013, http://www.w3.org/TR/2013/REC-prov-constraints-20130430/.

[30] T. D. Nies and S. Coppens, "PROV-dictionary: Modeling provenance for dictionary data structures," W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-dictionary-20130430/.

[31] Y. Gil and S. Miles, *PROV Model Primer*, W3C Working Group, Apr. 2013, 30. [Online]. Available: https://www.w3.org/TR/prov-primer

[32] H. Hua, S. Zednik, and C. Tilmes, “PROV-xml: The PROV xml schema,” W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-xml-20130430/.

[33] P. Groth and G. Klyne, “PROV-aq: Provenance access and query,” W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-aq-20130430/.

[34] L. Moreau and P. Missier, “PROV-n: The provenance notation,” W3C, W3C Recommendation, Apr. 2013, http://www.w3.org/TR/2013/REC-prov-n-20130430/.

[35] J. Cheney, “Semantics of the PROV data model,” W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-sem-20130430/.

[36] K. Eckert and D. Garijo, “Dublin core to PROV mapping,” W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-dc-20130430/.

[37] T. Lebo and L. Moreau, “Linking across provenance bundles,” W3C, W3C Note, Apr. 2013, http://www.w3.org/TR/2013/NOTE-prov-links-20130430/.

[38] C. Tilmes, P. Fox, X. Ma, D. L. McGuinness, A. P. Privette, A. Smith, A. Waple, S. Zednik, and J. G. Zheng, *Provenance Representation in the Global Change Information System (GCIS)*, ser. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, June 2012, vol. 7525, ch. Provenance and Annotation of Data and Processes, pp. 246–248. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34222-6_28

[39] X. Ma, P. Fox, C. Tilmes, K. Jacobs, and A. Waple, “Capturing provenance of global change information,” *Nature Clim. Change*, vol. 4, no. 6, pp. 409–413, Jun 2014, commentary. [Online]. Available: http://dx.doi.org/10.1038/nclimate2141

[40] I. Suriarachchi, Q. G. Zhou, and B. Plale, “Komadu: A capture and visualization system for scientific data provenance,” *Journal of Open Research Software*, vol. 3, no. 1, mar 2015. [Online]. Available: http://dx.doi.org/10.5334/jors.bq

[41] P. Ciccarese, S. Soiland-Reyes, K. Belhajjame, A. J. Gray, C. Goble, and T. Clark, “Pav ontology: provenance, authoring and versioning,” *Journal of Biomedical Semantics*, vol. 4, no. 1, p. 37, 2013. [Online]. Available: http://dx.doi.org/10.1186/2041-1480-4-37

[42] (2012, Jun.) Dcmi metadata terms. DCMI Usage Board. Accessed: February 8, 2017. [Online]. Available: http://dublincore.org/documents/2012/06/14/dcmi-terms/

[43] Updateaction. Schema.org. Accessed: January 19, 2017. [Online]. Available: http://schema.org/UpdateAction

[44] Replaceaction. Schema.org. Accessed: January 19, 2017. [Online]. Available: http://schema.org/ReplaceAction

[45] Addaction. Schema.org. Accessed: January 19, 2017. [Online]. Available: http://schema.org/AddAction

[46] Deleteaction. Schema.org. Accessed: January 19, 2017. [Online]. Available: http://schema.org/DeleteAction

[47] M. D. Flouris, "Clotho: Transparent data versioning at the block i/o level," in *In Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, 2004, pp. 315–328.

[48] R. Rantzau, C. Constantinescu, U. Heinkel, and H. Meinecke, "Champagne: Data change propagation for heterogeneous information systems," in *In: Proceedings of the International Conference on Very Large Databases (VLDB), Demonstration Paper, Hong Kong*, 2002.

[49] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo, "Version management of xml documents," in *Selected Papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases.* London, UK, UK: Springer-Verlag, 2001, pp. 184–200. [Online]. Available: http://dl.acm.org/citation.cfm?id=646544.696357

[50] A. Stuckenholz, "Component evolution and versioning state of the art," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 1, pp. 7–, Jan. 2005. [Online]. Available: http://doi.acm.org/10.1145/1039174.1039197

[51] J. Dijkstra, *On complex objects and versioning in complex environments.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 13–23. [Online]. Available: http://dx.doi.org/10.1007/BFb0024353

[52] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum, "A time machine for text search," in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '07. New York, NY, USA: ACM, 2007, pp. 519–526. [Online]. Available: http://doi.acm.org/10.1145/1277741.1277831

[53] S. Lyons, "Persistent identification of electronic documents and the future of footnotes," *Law Library Journal*, vol. 97, pp. 681–694, 2005.

[54] R. E. Duerr, R. R. Downs, C. Tilmes, B. Barkstrom, W. C. Lenhardt, J. Glassy, L. E. Bermudez, and P. Slaughter, "On the utility of identification schemes for digital earth science data: an assessment and recommendations,"

Earth Science Informatics, vol. 4, no. 3, p. 139, 2011. [Online]. Available: http://dx.doi.org/10.1007/s12145-011-0083-6

[55] B. R. Barkstrom, T. H. Hinke, S. Gavali, W. Smith, W. J. Seufzer, C. Hu, and D. E. Cordner, "Distributed generation of nasa earth science data products," *Journal of Grid Computing*, vol. 1, no. 2, pp. 101–116, 2003. [Online]. Available: http://dx.doi.org/10.1023/B:GRID.0000024069.33399.ee

[56] Data versioning. Australian National Data Service. Accessed: June 9, 2017. [Online]. Available: http://www.ands.org.au/working-with-data/data-management/data-versioning

[57] B. R. Barkstrom and J. J. Bates, "Digital library issues arising from earth science data," 2006.

[58] S. Payette and T. Staples, *The Mellon Fedora Project Digital Library Architecture Meets XML and Web Services.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 406–421. [Online]. Available: http://dx.doi.org/10.1007/3-540-45747-X_30

[59] W. F. Tichy, "Rcsa system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[60] P. Cederqvist, R. Pesch *et al.*, *Version management with CVS.* Network Theory Ltd., 2002.

[61] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 23–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=942800.943568

[62] P. Klahold, G. Schlageter, and W. Wilkes, "A general model for version management in databases," in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 319–327. [Online]. Available: http://dl.acm.org/citation.cfm?id=645913.671314

[63] J. F. Roddick, "A model for schema versioning in temporal database systems," *Australian Computer Science Communications*, vol. 18, pp. 446–452, 1996.

[64] P. Vassiliadis, M. Bouzeghoub, and C. Quix, *Towards Quality-Oriented Data Warehouse Usage and Evolution.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 164–179. [Online]. Available: http://dx.doi.org/10.1007/3-540-48738-7_13

[65] S. Proell and A. Rauber, "Scalable data citation in dynamic large databases: Model and reference implementation," in *IEEE International Conference on Big Data 2013 (IEEE BigData 2013)*, 10 2013.

[66] S. Pröll and A. Rauber, "Citable by design - A model for making data in dynamic environments citable," in *DATA 2013 - Proceedings of the 2nd International Conference on Data Technologies and Applications, Reykjavík, Iceland, 29 - 31 July, 2013*, 2013, pp. 206–210. [Online]. Available: http://dx.doi.org/10.5220/0004589102060210

[67] M. Helfert, C. Francalanci, and J. Filipe, Eds., *DATA 2013 - Proceedings of the 2nd International Conference on Data Technologies and Applications, Reykjavík, Iceland, 29 - 31 July, 2013.* SciTePress, 2013.

[68] K. Holtman, "CMS Data Grid System Overview and Requirements," CERN, Geneva, Tech. Rep. CMS-NOTE-2001-037, Jul 2001. [Online]. Available: http://cds.cern.ch/record/687353

[69] J. Kovse and T. Härder, "V-grid-a versioning services framework for the grid," in *Berliner XML Tage*, 2003.

[70] C. Ochs, Y. Perl, J. Geller, M. Haendel, M. Brush, S. Arabandi, and S. Tu, "Summarizing and visualizing structural changes during the evolution of biomedical ontologies using a diff abstraction network," *J. of Biomedical Informatics*, vol. 56, no. C, pp. 127–144, Aug. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jbi.2015.05.018

[71] M. Hartung, A. Gro, and E. Rahm, "Contodiff: generation of complex evolution mappings for life science ontologies," *Journal of Biomedical Informatics*, vol. 46, no. 1, pp. 15 – 32, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1532046412000627

[72] M. Klein and D. Fensel, "Ontology versioning on the semantic web," in *Stanford University*, 2001, pp. 75–91.

[73] C. Hauptmann, M. Brocco, and W. Wörndl, "Scalable semantic version control for linked data management," in *2nd Workshop on Linked Data Quality (LDQ)*, ser. CEUR Workshop Proceedings, A. Rula, A. Zaveri, M. Knuth, and D. Kontokostas, Eds., no. 1376, Aachen, 2015, accessed: February 21, 2017. [Online]. Available: http://ceur-ws.org/Vol-1376

[74] A. Rula, A. Zaveri, M. Knuth, and D. Kontokostas, Eds., *Proceedings of the 2nd Workshop on Linked Data Quality (LDQ)*, ser. CEUR Workshop Proceedings, no. 1376, Aachen, 2015. [Online]. Available: http://ceur-ws.org/Vol-1376/

[75] R. Bose and J. Frew, "Lineage retrieval for scientific data processing: A survey," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 1–28, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1057977.1057978

[76] S. Burrows, "A review of electronic journal acquisition, management, and use in health sciences libraries," *Journal of the Medical Library Association*, vol. 94, no. 1, pp. 67–74, 01 2006, copyright - Copyright Medical Library Association Jan 2006; Document feature - Graphs; Tables; ; Last updated - 2016-11-09. [Online]. Available: http://search.proquest.com/docview/203517273?accountid=28525

[77] "Common questions: Ubuntu release and version numbers," Canonical Ltd., accessed: December 12, 2016. [Online]. Available: https://help.ubuntu.com/community/CommonQuestions##Ubuntu%20Releases%20and%20Version%20Numbers

[78] S. McCarron, I. Herman, B. Adida, and M. Birbeck, "RDFa core 1.1 - third edition," W3C, W3C Recommendation, Mar. 2015, http://www.w3.org/TR/2015/REC-rdfa-core-20150317/.

[79] M. Sporny, I. Herman, B. Adida, and M. Birbeck, "RDFa 1.1 primer - third edition," W3C, W3C Note, Mar. 2015, http://www.w3.org/TR/2015/NOTE-rdfa-primer-20150317/.

[80] C. Bizer, K. Eckert, R. Meusel, H. Mühleisen, M. Schuhmacher, and J. Völker, *Deployment of RDFa, Microdata, and Microformats on the Web – A Quantitative Analysis.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 17–32. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41338-4_2

[81] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindstrom. (2017, Dec.) Json-ld 1.1. W3C. Accessed: June 7, 2017. [Online]. Available: https://json-ld.org/spec/latest/json-ld/

[82] M. Bouzeghoub and V. Peralta, "A framework for analysis of data freshness," in *Proceedings of the 2004 International Workshop on Information Quality in Information Systems*, ser. IQIS '04. New York, NY, USA: ACM, 2004, pp. 59–67. [Online]. Available: http://doi.acm.org/10.1145/1012453.1012464

[83] C. Tilmes, Y. Yesha, and M. Halem, "Distinguishing provenance equivalence of earth science data," *Procedia Computer Science*, vol. 4, pp. 548 – 557, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050911001153

[84] E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo, "Approximated summarization of data provenance," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge*

Management, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 483–492. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806429

[85] A. Hliaoutakis, G. Varelas, E. Voutsakis, E. G. M. Petrakis, and E. Milios, "Information retrieval by semantic similarity," in *Intern. Journal on Semantic Web and Information Systems (IJSWIS), 3(3):5573, July/Sept. 2006. Special Issue of Multimedia Semantics*, 2006.

[86] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Provenance-based object storage prediction scheme for scientific big data applications," in *Big Data (Big Data), 2014 IEEE International Conference on.* IEEE, 2014, pp. 271–280.

[87] B. Cao, Y. Li, and J. Yin, "Measuring similarity between gaphs based on the levenshtein distance," *Applied Mathematics & Information Sciences*, vol. 7, no. 1L, pp. 169–175, 2013.

[88] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Analysis and Applications*, vol. 13, no. 1, pp. 113–129, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10044-008-0141-y

[89] W. Goddard and H. C. Swart, "Distances between graphs under edge operations," *Discrete Math.*, vol. 161, no. 1-3, pp. 121–132, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1016/0012-365X(95)00073-6

[90] Y. Ma, M. Shi, and J. Wei, "Cost and accuracy aware scientific workflow retrieval based on distance measure," *Information Sciences*, vol. 314, no. C, pp. 1–13, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2015.03.055

[91] W. C. Tan, "Research problems in data provenance." *IEEE Data Eng. Bull.*, vol. 27, no. 4, pp. 45–52, 2004.

[92] B. Polyak, E. Prasolov, I. Tolstikhin, L. Yakovlev, A. Ioffe, O. Kikvadze, O. Vereina, and M. Vetrina, "Noble gas isotope abundances in terrestrial fluids," 2015. [Online]. Available: https://info.deepcarbon.net/vivo/display/n6225

[93] S. Morrison, R. Downs, J. Golden, A. Pires, P. Fox, X. Ma, S. Zednik, A. Eleish, A. Prabhu, D. Hummer, C. Liu, M. Meyer, J. Ralph, G. Hystad, and R. Hazen, "Exploiting mineral data: applications to the diversity, distribution, and social networks of copper mineral," in *AGU Fall Meeting*, 2016.

[94] K. Elder, D. Cline, G. E. Liston, and R. Armstrong, "Nasa cold land processes experiment (clpx 2002/03): Field measurements of snowpack properties and soil moisture," *Journal of Hydrometeorology*, vol. 10, no. 1, pp. 320–329, 2009. [Online]. Available: https://doi.org/10.1175/2008JHM877.1

[95] M. A. Parsons, M. J. Brodzik, and N. J. Rutter, “Data management for the cold land processes experiment: improving hydrological science,” *Hydrological Processes*, vol. 18, no. 18, pp. 3637–3653, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/hyp.5801

[96] Z. B. Miled, S. Sikkupparbathyam, O. Bukhres, K. Nagendra, E. Lynch, M. Areal, L. Olsen, C. Gokey, D. Kendig, T. Northcutt, R. Cordova, G. Major, and N. Savage, “Global change master directory: Object-oriented active asynchronous transaction management in a federated environment using data agents,” in *Proceedings of the 2001 ACM Symposium on Applied Computing*, ser. SAC ’01. New York, NY, USA: ACM, 2001, pp. 207–214. [Online]. Available: http://doi.acm.org/10.1145/372202.372324

[97] “Keyword faq,” Earthdata, 2016, accessed: December 12, 2016. [Online]. Available: https://wiki.earthdata.nasa.gov/display/CMR/Keyword+FAQ

[98] A. Miles and S. Bechhofer, “SKOS simple knowledge organization system reference,” W3C, W3C Recommendation, Aug. 2009, http://www.w3.org/TR/2009/REC-skos-reference-20090818/.

[99] T. Stevens, “Nasa gcmd keyword version 8.4 released,” Aug. 2016, accessed: February 10, 2017. [Online]. Available: https://wiki.earthdata.nasa.gov/display/CMR/NASA+GCMD+Keywords+Version+8.4+Released

[100] Marine biodiversity virtual laboratory. Accessed: September 28, 2016. [Online]. Available: https://tw.rpi.edu/web/project/MBVL

[101] Global Change Master Directory (GCMD), *Global Change Master Directory (GCMD) Keyword Governance and Community Guide Document*, version 1.0 ed., National Aeronautics and Space Administration (NASA), Aug. 2016, accessed: June 10, 2018. [Online]. Available: https://cdn.earthdata.nasa.gov/conduit/upload/5182/KeywordsCommunityGuide_Baseline_v1_SIGNED_FINAL.pdf

[102] UCAR/NCAR - Earth Observing Laboratory. (2018) About eol. UCAR/NCAR - Earth Observing Laboratory. Accessed: June 10, 2018. [Online]. Available: https://www.eol.ucar.edu/about-eol

APPENDIX A
NOBLE GAS CHANGE LOG GENERATOR VERSION 1 TO 2

```python
from os.path import join, dirname, abspath, isfile
from os import sep as separator
import xlrd, sys, json
import glob
import re


def index_convert(index1):
        if index1 < 17:
                return index1
        elif index1 < 24:
                return index1+1
        elif index1 < 26:
                return 26+(3*(index1-24))
        elif index1 < 28:
                return 44+(3*(index1-26))
        elif index1 < 32:
                return 53+(9*(index1-28))
        elif index1 < 36:
                return 88+(2*(index1-32))
        elif index1 < 38:
                return 95+(4*(index1-36))
        elif index1 < 41:
                return 102+(2*(index1-38))
        elif index1 < 43:
                return 112+(2*(index1-41))
        elif index1 == 43:
                return 172
        elif index1 < 50:
                return 174+(3*(index1-44))
```

```
        elif index1 < 54:
                return 191+(index1-50)
        else:
                print 'Error: Out of bounds'
                return -1

def test_alignment():
        for i in range(0, 54):
                print 'version2: {:5} version1: {:5}'.format(i,
    ↪ index_convert(i))

def compare_print(mode, key, val1, val2, v1_file, v1_index = 0,
    ↪ v2_index = 0, changelog = None):
        if changelog:
                if mode == 'r':
                        out = u'''        <tr  about="Change{}{}"
    ↪ typeof="vo:ModifyChange">
          <td align="right" rev="vo:Undergoes"
    ↪ resource="v1:Attribute{}{}v1" typeof="vo:Attribute">{:2}({})</td>
          <td property="vo:resultsIn" resource="v2:Attribute{}{}v2"
    ↪ typeof="vo:Attribute">{:2}</td>
          <td>{:>10}</td>
          <td>{:>10}</td>
          <span about="Version1" property="vo:hasAttribute"
    ↪ resource="v1:Attribute{}{}v1"></span>
          <span about="Version2" property="vo:hasAttribute"
    ↪ resource="v2:Attribute{}{}v2"></span>
        </tr>\n'''.format(key, v2_index, key, v1_index, v1_index,
    ↪ v1_file, key, v2_index, v2_index, val1, val2, key, v1_index,
    ↪ key, v2_index)
                elif mode == 'j':
                        out = u'''        <tr  id="ModifyChange{}{}">
          <td align="right">{:2}({})</td>
          <td>{:2}</td>
          <td>{:>10}</td>
          <td>{:>10}</td>
```

```
          <script type="application/ld+json">\n'''.format(key, v2_index, v1_index, v1_file, v2_index, val1, val2)
                elif mode == 't':
                        out = u"{:2}({})\t{:2}\t{:>10}\t{:>10}\n".format(v1_index, v1_file, v2_index, val1, val2)
                elif mode == 'u':
                        out = u"""<http://example.com/NG/Version1> vo:hasAttribute <http://example.com/NG/Version1/%s> ;
        vo:hasAttribute <http://example.com/NG/Version1/Column%i> .
<http://example.com/NG/Version1/%s> a vo:Attribute ;
        vo:undergoes <http://example.com/Changelog#ModifyChange%s%i> .
<http://example.com/NG/Version1/Column%i> a vo:Attribute ;
        vo:undergoes <http://example.com/Changelog#ModifyChange%s%i> .
<http://example.com/Changelog#ModifyChange%s%i> a vo:ModifyChange ;
        vo:resultsIn <http://example.com/NG/Version2/%s> ;
        vo:resultsIn <http://example.com/NG/Version2/Column%i> .
<http://example.com/NG/Version2> vo:hasAttribute <http://example.com/NG/Version2/%s> ;
        vo:hasAttribute <http://example.com/NG/Version2/Column%i> .

"""%(key, v1_index, key, key, v2_index, v1_index, key, v2_index, key, v2_index, key, v2_index, key, v2_index)
                changelog.write(out.encode('utf8'))
                if mode == 'j':
                        json1 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v1/Attribute", key, str(v1_index)]) ,
"label":key ,
"undergoes":"".join([host, "ModifyChange", key, str(v2_index)]) ,
"@reverse" :    { "hasAttribute" : "Version1" }
}
                        json2 = {
```

```
"@context":context,
"@type":"vo:ModifyChange",
"@id":"".join([host, "ModifyChange", key, str(v2_index)]) ,
"resultsIn":"".join(["http://ngdb.com/v2/Attribute", key, str(v2_index)])
}
                        json3 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v2/Attribute", key, str(v2_index)]) ,
"label":key ,
"@reverse" :    { "hasAttribute" : "Version2" }
}
                        json.dump([json1, json2, json3], changelog, indent=4, sort_keys=True)
                        changelog.write('''
          </script>
        </tr>
''')
        else:
                print '{:5}  version1: {:10} version2: {:10}'.format(key, val1, val2)

labels = {17:"SAMPLING - DEPTH - >,<",
                  25:"[He] - ppm - >,<", 27:"[He] - ppm - err", 28:"[He] - mkcc/ - >,<", 30:"[He] - mkcc/ - err", 31:"[He] - mol/ - >,<", 32:"[He] - mol/ - L H2O", 33:"[He] - mol/ - err",
                  34:"[He+Ne] - ppm - >,<", 35:"[He+Ne] - ppm", 36:"[He+Ne] - ppm - err", 37:"[He+Ne] - mkcc/ - >,<", 38:"[He+Ne] - mkcc/ - g H2O", 39:"[He+Ne] - mkcc/ - err", 40:"[He+Ne] - mol/ - >,<", 41:"[He+Ne] - mol/ - L H2O", 42:"[He+Ne] - mol/ - err",
                  43:"[Ne] - ppm - >,<", 45:"[Ne] - ppm - err", 46:"[Ne] - mkcc/ - >,<", 48:"[Ne] - mkcc/ - err", 49:"[Ne] - mol/ - >,<", 50:"[Ne] - mol/ - L H2O", 51:"[Ne] - mol/ - err",
```

```
                    52:"[20Ne] - ppm - >,<", 54:"[20Ne] - ppm - err", 55:"[20Ne] - mkcc/ - >,<", 56:"[20Ne] - mkcc/ - g H2O", 57:"[20Ne] - mkcc/ - err", 58:"[20Ne] - mol/ - >,<", 59:"[20Ne] - mol/ - L H2O", 60:"[20Ne] - mol/ - err",
                    61:"[Ar] - ppm - >,<", 63:"[Ar] - ppm - err", 64:"[Ar] - mkcc/ - >,<", 65:"[Ar] - mkcc/ - g H2O", 66:"[Ar] - mkcc/ - err", 67:"[Ar] - mol/ - >,<", 68:"[Ar] - mol/ - L H2O", 69:"[Ar] - mol/ err",
                     70:"[Kr] - ppm - >,<", 72:"[Kr] - ppm - err", 73:"[Kr] - mkcc/ - >,<", 74:"[Kr] - mkcc/ - g H2O", 75:"[Kr] - mkcc/ - err", 76:"[Kr] - mol/ - >,<", 77:"[Kr] - mol/ - L H2O", 78:"[Kr] - mol/ err",
                    79:"[Xe] - ppm - >,<", 81:"[Xe] - ppm - err", 82:"[Xe] - mkcc/ - >,<", 83:"[Xe] - mkcc/ - g H2O", 84:"[Xe] - mkcc/ - err", 85:"[Xe] - mol/ - >,<", 86:"[Xe] - mol/ - L H2O", 87:"[Xe] - mol/ err",
                    89:"3He/4He - (R/Ra)me - err", 91:"3He/4He - (R/Ra)corr - err", 93:"3He/4He - Rme - E-8 - err", 96:"3He/4He - Rcorr - E-8 - err", 97:"Rank",
                    98:"He/Ne - >,<", 100:"He/Ne - >,<", 101:"4He/20Ne - >,<", 103:"4He/20Ne - err", 105:"20Ne/22Ne - err", 107:"21Ne/22Ne - (xE-2) - err", 108:"21Ne/20Ne", 109:"21Ne/20Ne - err",
                    110:"22Ne/20Ne", 111:"22Ne/20Ne - err", 113:"38Ar/36Ar - err", 115:"40Ar/36Ar - err", 116:"delta(40Ar)rad", 117:"delta(40Ar)rad - err",
                    118:"He/Ar - He/ - /Ar(air) - >,<", 119:"He/Ar - He/ - /Ar(air)", 120:"He/Ar - He/ - /Ar(air) - err", 121:"He/Ar - 4He/ - /36Ar - >,<", 122:"He/Ar - 4He/ - /36Ar", 123:"He/Ar - 4He/ - /36Ar - err",
                    124:"He/Ar - 4He/ - /40Ar(air) - >,<", 125:"He/Ar - 4He/ - /40Ar(air)", 126:"He/Ar - 4He/ - /40Ar(air) - err",
                    127:"f(He)=(He/Ar)s/(He/Ar)air - >,<", 128:"f(He)=(He/Ar)s/(He/Ar)air", 129:"f(He)=(He/Ar)s/(He/Ar)air - err",
```

```
                    130:"Ne/Ar - Ne/ - /Ar(air) - >,<", 131:"Ne/Ar - Ne/ - /Ar(air)", 132:"Ne/Ar - Ne/ - /Ar(air) - err", 133:"Ne/Ar - 20Ne/ - /36Ar - >,<", 134:"Ne/Ar - 20Ne/ - /36Ar", 135:"Ne/Ar - 20Ne/ - /36Ar - err",
                    136:"Ne/Ar - 20Ne/ - /40Ar(air) - >,<", 137:"Ne/Ar - 20Ne/ - /40Ar(air)", 138:"Ne/Ar - 20Ne/ - /40Ar(air) - err", 139:"Ne/Ar - 22Ne/ - /36Ar - >,<", 140:"Ne/Ar - 22Ne/ - /36Ar", 141:"Ne/Ar - 22Ne/ - /36Ar - err",
                    142:"Ne/Ar - 22Ne/ - /40Ar(air) - >,<", 143:"Ne/Ar - 22Ne/ - /40Ar(air)", 144:"Ne/Ar - 22Ne/ - /40Ar(air) - err",
                    145:"f(Ne)=(Ne/Ar)s/(Ne/Ar)air - >,<", 146:"f(Ne)=(Ne/Ar)s/(Ne/Ar)air", 147:"f(Ne)=(Ne/Ar)s/(Ne/Ar)air - err",
                    148:"Kr/Ar - Kr/ - /Ar(air) - >,<", 149:"Kr/Ar - Kr/ - /Ar(air)", 150:"Kr/Ar - Kr/ - /Ar(air) - err", 151:"Kr/Ar - 84Kr/ - /36Ar - >,<", 152:"Kr/Ar - 84Kr/ - /36Ar", 153:"Kr/Ar - 84Kr/ - /36Ar - err",
                    154:"Kr/Ar - 84Kr/ - /40Ar(air) - >,<", 155:"Kr/Ar - 84Kr/ - /40Ar(air)", 156:"Kr/Ar - 84Kr/ - /40Ar(air) - err",
                    157:"f(Kr)=(Kr/Ar)s/(Kr/Ar)air - >,<",158:"f(Kr)=(Kr/Ar)s/(Kr/Ar)air", 159:"f(Kr)=(Kr/Ar)s/(Kr/Ar)air - err",
                    160:"Xe/Ar - Xe/ - /Ar(air) - >,<", 161:"Xe/Ar - Xe/ - /Ar(air)", 162:"Xe/Ar - Xe/ - /Ar(air) - err", 163:"Xe/Ar - 132Xe/ - /36Ar - >,<", 164:"Xe/Ar - 132Xe/ - /36Ar", 165:"Xe/Ar - 132Xe/ - /36Ar - err",
                    166:"Xe/Ar - 132Xe/ - /40Ar(air) - >,<", 167:"Xe/Ar - 132Xe/ - /40Ar(air)", 168:"Xe/Ar - 132Xe/ - /40Ar(air) - err",
                    169:"f(Xe)=(Xe/Ar)s/(Xe/Ar)air - >,<", 170:"f(Xe)=(Xe/Ar)s/(Xe/Ar)air", 171:"f(Xe)=(Xe/Ar)s/(Xe/Ar)air - err",
                    173:"H2 - >,<", 175:"H2 - ppm - err", 176:"O2 - >,<", 178:"O2 - ppm - err", 179:"N2 - >,<", 181:"N2 - ppm - err", 182:"CO2 - >,<", 184:"CO2 - ppm - err", 185:"CH4 - >,<", 187:"CH4 - ppm - err",
```

```
                    188:"H2S - >,<", 190:"H2S - ppm - err"}

context = "https://orion.tw.rpi.edu/~blee/provdist/GCMD/VO.jsonld"
host = "http://orion.tw.rpi.edu/~blee/provdist/NobleGas/changelog_json.html#"
#test_alignment()


#print v2_row[0].value
#print indicator_map[v2_row[0].value]

#v1_workbook = xlrd.open_workbook(v1_file)
#v1_sheet = v1_workbook.sheet_by_index(0)
#v1_row = v1_sheet.row(4)

def write_modify(r1, r2, workbook, f_out, mode):
        if mode == 'r':
                out = u'''  <div about="Version1" rel="vo:hasAttribute">
    <div resource="v2:%s" typeof="vo:Attribute">
      <span style="font-weight:bold" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span>
      <table rel="vo:Undergoes">
'''%(r2[0].value, r2[0].value)
        elif mode == 'j':
                out = u'''
    <div about="v2:%s">
      <span style="font-weight:bold" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span>
      <table>
'''%(r2[0].value, r2[0].value)
        elif mode == 't':
                out = u"%s\n"%(r2[0].value)
        elif mode == 'u':
                out = u""

```

```
        if mode == 'r' or mode == 'j':
                out = out+'''        <tr>
          <th>Column v1</th>
          <th>Column v2</th>
          <th>Version 1</th>
          <th>Version 2</th>
        </tr>\n'''
        elif mode == 't':
                out = out+"Column v1\tColumn v2\tVersion 1\tVersion 2\n"
        f_out.write(out.encode('utf8'))
                #print '# Searching...'
                #print '# Comparing...'
        for i in range(0,54):
                if r2[i].value != r1[index_convert(i)].value:
                        #compare_print(j, v1_row[index_convert(j)].value, v2_row[j].value)
                        compare_print(mode, r2[0].value, r1[index_convert(i)].value, r2[i].value, workbook.split('/')[-1], index_convert(i), i, f_out)
        if mode == 'r' or mode == 'j':
                f_out.write('  </table></div><br>\n')
        elif mode == 't' or mode == 'u':
                f_out.write("\n")

def write_removed(v2, col, row, f_out, mode):
        if mode == 'r' or mode == 'j':
                f_out.write('''
      <h3>Columns invalidated by %s</h3>
      <table about="Version2">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nColumns invalidated by %s\n"%(v2.split('/')[-1]))

        print "Removed Column"
```

```
        for i in col:
                v1_value = labels.get(i, "")
                if mode == 'r':
                        out = u'''        <tr resource="InvlidateChange%i" rev="vo:invalidatedBy" typeof="vo:InvalidateChange">
                <td resource="Attribute%i" rev="vo:Undergoes" typeof="vo:Attribute">%i</td>
          <td about="Attribute%i" property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
          <span about="Version1" property="vo:hasAttribute" resource="Attribute%i"/>
        </tr>
'''%(i, i, i, i, v1_value, i)
                elif mode == 'j':
                        out = u'''        <tr id="InvlidateChange%i" about="InvlidateChange%i">
          <td>%i</td>
          <td>%s</td>
          <script type="application/ld+json">
'''%(i, i, i, v1_value)
                elif mode == 't':
                        out = u"%i\t%s\n"%(i, v1_value)
                elif mode == 'u':
                        out = u"""<http://example.com/NG/Version1> vo:hasAttribute <http://example.com/NG/Version1/Column%s> .
<http://example.com/NG/Version1/%s> vo:undergoes <http://example.com/Changelog#InvalidateChange%i> .
<http://example.com/Changelog#InvalidateChange%i> a vo:InvalidateChange ;
        vo:invalidatedBy <http://example.com/NG/Version2> .

"""%(i, i, i, i)
                f_out.write(out.encode('utf8'))
                if mode == 'j':
                        json1 = {
```

```
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v1/Attribute", str(i)]) ,
"label": v1_value,
"undergoes":"".join([host, "InvalidateChange", str(i)]) ,
"@reverse" :    { "hasAttribute" : "Version1" }
}
                        json2 = {
"@context":context,
"@type":"vo:InvalidateChange" ,
"@id": "".join([host, "InvalidateChange", str(i)]) ,
"invalidatedBy"  :   "Version2"
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
''')


        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>
      <h3>Rows invalidated by %s</h3>
      <table about="Version2">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nRows invalidated by %s\n"%(v2.split('/')[-1]))
        elif mode == 'u':
                f_out.write("\n")

        print "Removed Row"
        workbook_name = ''
        for i, j in sorted(row, key=lambda x: x[0]):
                if workbook_name != j:
```

```
                    workbook_name = j
                    v1_workbook = xlrd.open_workbook(workbook_name)
                    v1_sheet = v1_workbook.sheet_by_index(0)
                    v1_col = v1_sheet.col(0)
                    v1_col = [k.value for k in v1_col]
                v1_index = v1_col.index(i)
                if mode == 'r':
                    out = u'''        <tr resource="InvlidateChange%i" rev="vo:invalidatedBy" typeof="vo:InvalidateChange">
            <td resource="Attribute%i" rev="vo:Undergoes" typeof="vo:Attribute">%i(%s)</td>
            <td about="Attribute%i" property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
            <span about="Version1" property="vo:hasAttribute" resource="Attribute%i"/>
          </tr>
'''%(v1_index, v1_index, v1_index, workbook_name.split('/')[-1], v1_index, i, v1_index)
                elif mode == 'j':
                    out = u'''        <tr id="InvlidateChange%i" about="InvlidateChange%i">
            <td>%i(%s)</td>
            <td>%s</td>
            <script type="application/ld+json">
'''%(v1_index, v1_index, v1_index, workbook_name.split('/')[-1], i)
                elif mode == 't':
                    out = u"%i(%s)\t%s\n"%(v1_index, workbook_name.split('/')[-1], i)
                elif mode == 'u':
                    out = u"""<http://example.com/NG/Version1> vo:hasAttribute <http://example.com/NG/Version1/%s> .
<http://example.com/NG/Version1/%s> vo:undergoes <http://example.com/Changelog#InvalidateChange%s> .
```

```
<http://example.com/Changelog#InvalidateChange%s> a vo:InvalidateChange ;
        vo:invalidatedBy <http://example.com/NG/Version2> .

"""%(i, i, i, i)
                f_out.write(out.encode('utf8'))
                if mode == 'j':
                        json1 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v1/Attribute", str(i)]) ,
"label": str(i),
"undergoes":"".join([host, "InvalidateChange", str(i)]) ,
"@reverse" :    { "hasAttribute" : "Version1" }
}
                        json2 = {
"@context":context,
"@type":"vo:InvalidateChange" ,
"@id": "".join([host, "InvalidateChange", str(i)]) ,
"invalidatedBy"  :   "Version2"
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
''')
        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>

''')
        elif mode == 't' or mode == 'u':
                f_out.write("\n")


def write_added(v2, col, row, f_out, mode):
```

```
        if mode == 'r' or mode == 'j':
                f_out.write('''
      <h3>Columns added by %s</h3>
      <table about="Version1" rel="vo:absentFrom">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nColumns added by %s\n\n"%(v2.split('/')[-1]))

        print "Added Column"
        for i in col:
                print i#, v2_value
                if mode == 'r':
                        f_out.write('''        <tr about="AddChange%i" typeof="vo:AddChange">
          <td property="vo:resultsIn" resource="Attribute%i" typeof="vo:Attribute">%i</td>
          <td about="Attribute%i" property="http://www.w3.org/2000/01/rdf-schema#label"></td>
          <span about="Version2" property="vo:hasAttribute" resource="Attribute%i"/>
        </tr>
'''%(i, i, i, i, i))
                elif mode == 'j':
                        f_out.write('''        <tr id="AddChange%i" about="v2:Attribute%i">
          <td>%i</td>
          <td></td>
          <script type="application/ld+json">
'''%(i, i, i))
                        json1 = {
"@context":context,
"@type":"vo:AddChange" ,
"@id": "".join([host, "AddChange", str(i)]) ,
"resultsIn" :   "".join([ "http://ngdb.com/v2/Attribute", str(i)]),
"@reverse"  :   { "absentFrom": "Version1" }
```

```
}
                        json2 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v2/Attribute", str(i)]) ,
"label":"" ,
"@reverse" :    { "hasAttribute" : "Version2" }
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
''')
                elif mode == 't':
                        f_out.write("%i\t\n"%(i))
                elif mode == 'u':
                        f_out.write("""<http://example.com/NG/Version1> vo:absentFrom <http://example.com/Changelog#AddChange%i> .
<http://example.com/Changelog#AddChange%i> a vo:AddChange ;
        vo:resultsIn <http://example.com/NG/Version2/Column%s> .
<http://example.com/NG/Version2> vo:hasAttribute <http://example.com/NG/Version2/Column%s> .

"""%(i, i, i, i))
        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>
              <h3>Rows added by %s</h3>
              <table about="Version1" rel="vo:absentFrom">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nRows added by %s\n\n"%(v2.split('/')[-1]))
        elif mode == 'u':
                f_out.write("\n")
```

```

        print "Added Row"
        for i, j in row:#i is the id, j is the file
                if mode == 'r':                        #print i, v2_sheet.cell(i,0).value
                        out = u'''        <tr about="AddChange%i" typeof="vo:AddChange">
          <td property="vo:resultsIn" resource="Attribute%i" typeof="vo:Attribute">%i</td>
          <td about="Attribute%i" property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
          <span about="Version2" property="vo:hasAttribute" resource="Attribute%i"/>
        </tr>
'''%(i, i, i, i, j, i)
                elif mode == 'j':
                        out = u'''        <tr id="AddChange%i" about="v2:Attribute%i">
          <td>%i</td>
          <td property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
          <script type="application/ld+json">
'''%(i, i, i, j)
                elif mode == 't':
                        out = u"%i\t%s\n"%(i, j)
                elif mode == 'u':
                        out = u"""<http://example.com/NG/Version1> vo:absentFrom <http://example.com/Changelog#AddChange%i> .
<http://example.com/Changelog#AddChange%i> a vo:AddChange ;
        vo:resultsIn <http://example.com/NG/Version2/%s> .
<http://example.com/NG/Version2> vo:hasAttribute <http://example.com/NG/Version2/%s> .

"""%(i, i, i, i)
                f_out.write(out.encode('utf8'))
                if mode == 'j':
```

```
                        json1 = {
"@context":context,
"@type":"vo:AddChange" ,
"@id": "".join([host, "AddChange", str(i)]) ,
"resultsIn" :   "".join([ "http://ngdb.com/v2/Attribute", str(i)]),
"@reverse"  :   { "absentFrom": "Version1" }
}
                        json2 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v2/Attribute", str(i)]) ,
"label": j ,
"@reverse" :    { "hasAttribute" : "Version2" }
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
''')

        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>
''')
        elif mode == 't' or mode == 'u':
                f_out.write("\n")

def write_header(f_out, mode):
        if mode == 'j' or mode == 'r':
                f_out.write('''<html>
  <head>
  </head>
  <body vocab="http://www.w3.org/nw/prov#" prefix="vo: https://orion.tw.rpi.edu/~blee/VersionOntology.owl# v1: http://ngdb.com/v1/ v2: http://ngdb.com/v2/">
''')
```

```
        if mode == 'j':
                f_out.write('''  <script type="application/ld+json">
''')
                json1 = {
"@context":context,
"@type":"vo:Version",
"@id":"Version1",
"label":"ngdbv1"
}
                json2 = {
"@context":context,
"@type":"vo:Version",
"@id":"Version2",
"label":"DB_final-55-7262_2015_03_08.xlsx"
}
                json.dump([json1,json2], f_out, indent=4, sort_keys=True)
                f_out.write("\n  </script>\n")
        if mode == 'u':
                f_out.write("""@prefix vo: <http://orion.tw.rpi.edu/~blee/VersionOntology.owl#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.com/NG/Version1> a vo:Version ;
        skos:prefLabel "ngdbv1" .

<http://example.com/NG/Version2> a vo:Version ;
        skos:prefLabel "DB_final-55-7262_2015_03_08.xlsx" .

""")

def write_footer(f_out, mode):
```

```
        if mode == 'r':
                f_out.write('</body>\n</html>')

def get_indicator_map(excel_files):
        indicator_map = {}
        for excel_file in excel_files:
                print 'Importing: ' + excel_file
                file_workbook = xlrd.open_workbook(excel_file)
                file_sheet = file_workbook.sheet_by_index(0)
                indicators = file_sheet.col(0)
                for i in range(4, file_sheet.nrows):
                        indicator_map[indicators[i].value] = excel_file
        return indicator_map

def compare(v1s, v2, fn_out, mode):
        indicator_map = get_indicator_map(v1s)
        i_keys = indicator_map.keys()
        v2_workbook = xlrd.open_workbook(v2)
        f_out = open(fn_out, 'w')

        v2_sheet = v2_workbook.sheet_by_index(0)
        v2_keys = [i.value for i in v2_sheet.col(0)]

        converted = [index_convert(i) for i in range(0,54)]
        new_col = [i for i in range(0, v2_sheet.ncols) if index_convert(i) == -1]
        new_row = [(i, v2_sheet.cell(i,0).value) for i in range(3, v2_sheet.nrows) if v2_sheet.cell(i,0).value not in i_keys]
        old_col = [i for i in range(0,194) if i not in converted]
        old_row = [(i, indicator_map.get(i, None)) for i in i_keys if i not in v2_keys]

        write_header(f_out, mode)
        write_added(v2, new_col, new_row, f_out, mode)
        write_removed(v2, old_col, old_row, f_out, mode)
```

```

        if mode == 'r' or mode == 'j':
                f_out.write('''
      <h3>Change Log</h3>
''')
        elif mode == 't':
                f_out.write("Change Log\n")

        workbook_name = ''
        for i in range(3,v2_sheet.nrows):
                v2_row = v2_sheet.row(i)
                #workbook_name = v1_file
                if v2_row[0].value in [j for i, j in new_row] or v2_row[0].value in [i for i, j in old_row]:
                        continue
                if workbook_name == indicator_map.get(v2_row[0].value, None):
                        pass
                else:
                        workbook_name = indicator_map.get(v2_row[0].value, None)
                        v1_workbook = xlrd.open_workbook(workbook_name)
                        v1_sheet = v1_workbook.sheet_by_index(0)
                        v1_col = v1_sheet.col(0)
                        v1_col = [j.value for j in v1_col]
                #print v2_row[0].value
                v1_index = v1_col.index(v2_row[0].value)
                v1_row = v1_sheet.row(v1_index)
                write_modify(v1_row, v2_row, workbook_name, f_out, mode)

        write_footer(f_out, mode)
        f_out.close()

if __name__ == "__main__":
```

```
        if '-json' in sys.argv:
                mode = 'j'
                out_name = 'changelog_json.html'
        elif '-rdfa' in sys.argv:
                mode = 'r'
                out_name = 'changelog_test.html'
        elif '-txt' in sys.argv:
                mode = 't'
                out_name = 'changelog.txt'
        elif '-ttl' in sys.argv:
                mode = 'u'
                out_name = 'changelog.ttl'

        v2_dir = join(separator, 'data', 'NGdata', 'v2')
        v1_dir = join(separator, 'data', 'NGdata', 'v1')

        excel_files = glob.glob("/data/NGdata/v1/DB_HE_6733.xlsx") #join(v1_dir, '*.xlsx'))

        v1_file = join(v1_dir, 'America_906.xlsx')
        v2_file = join(v2_dir, 'DB_final-55-7262_2015_03_08.xlsx')

        compare(excel_files, v2_file, out_name, mode)
```

APPENDIX B
NOBLE GAS CHANGE LOG GENERATOR VERSION 2 TO 3

```
from os.path import join, dirname, abspath, isfile
from os import sep as separator
import xlrd, sys, json
import glob
import re


def index_convert(index1):
        return index1

def test_alignment():
        for i in range(0, 54):
                print 'version2: {:5} version1: {:5}'.format(i, index_convert(i))

def compare_print(mode, key, val1, val2, v1_file, v1_index = 0, v2_index = 0, changelog = None):
        if changelog:
                if mode == 'r':
                        out = u'''       <tr  about="Change{}{}" typeof="vo:ModifyChange">
          <td align="right" rev="vo:Undergoes" resource="v2:Attribute{}{}v2" typeof="vo:Attribute">{:2}({})</td>
          <td property="vo:resultsIn" resource="v3:Attribute{}{}v3" typeof="vo:Attribute">{:2}</td>
          <td>{:>10}</td>
          <td>{:>10}</td>
          <span about="Version2" property="vo:hasAttribute" resource="v2:Attribute{}{}v2"></span>
```

```
            <span about="Version3" property="vo:hasAttribute" resource="v3:Attribute{}{}v3"></span>
        </tr>\n'''.format(key, v2_index, key, v1_index, v1_index, v1_file, key, v2_index, v2_index, val1, val2, key, v1_index, key, v2_index)
                elif mode == 'j':
                        out = u'''        <tr  id="ModifyChange{}{}">
            <td align="right">{:2}</td>
            <td>{:2}</td>
            <td>{:>10}</td>
            <td>{:>10}</td>
            <script type="application/ld+json">\n'''.format(key, v2_index, v1_index, v2_index, val1, val2)
                elif mode == 't':
                        out = u"{:2}\t{:2}\t{:>10}\t{:>10}\n".format(v1_index, v2_index, val1, val2)
                elif mode == 'u':
                        out = u"""<http://example.com/NG/Version2> vo:hasAttribute <http://example.com/NG/Version2/%s> ;
        vo:hasAttribute <http://example.com/NG/Version2/Column%i> .
<http://example.com/NG/Version2/%s> a vo:Attribute ;
        vo:undergoes <http://example.com/Changelog#ModifyChange%s%i> .
<http://example.com/NG/Version2/Column%i> a vo:Attribute ;
        vo:undergoes <http://example.com/Changelog#ModifyChange%s%i> .
<http://example.com/Changelog#ModifyChange%s%i> a vo:ModifyChange ;
        vo:resultsIn <http://example.com/NG/Version3/%s> ;
        vo:resultsIn <http://example.com/NG/Version3/Column%i> .
<http://example.com/NG/Version3> vo:hasAttribute <http://example.com/NG/Version3/%s> ;
        vo:hasAttribute <http://example.com/NG/Version3/Column%i> .

"""%(key, v1_index, key, key, v2_index, v1_index, key, v2_index, key, v2_index, key, v2_index, key, v2_index)
```

```
                changelog.write(out.encode('utf8'))
                if mode == 'j':
                        json1 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v2/Attribute", key, str(v1_index)]) ,
"label":key ,
"undergoes":"".join([host, "ModifyChange", key, str(v2_index)]) ,
"@reverse" :    { "hasAttribute" : "Version2" }
}
                        json2 = {
"@context":context,
"@type":"vo:ModifyChange",
"@id":"".join([host, "ModifyChange", key, str(v2_index)]) ,
"resultsIn":"".join(["http://ngdb.com/v3/Attribute", key, str(v2_index)])
}
                        json3 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v3/Attribute", key, str(v2_index)]) ,
"label":key ,
"@reverse" :    { "hasAttribute" : "Version3" }
}
                        json.dump([json1, json2, json3], changelog, indent=4, sort_keys=True)
                        changelog.write('''
          </script>
        </tr>
''')
        else:
                print '{:5}  version2: {:10} version3: {:10}'.format(key, val1, val2)

labels = {}

```

```
context = "https://orion.tw.rpi.edu/~blee/provdist/GCMD/VO.jsonld"
host = "http://orion.tw.rpi.edu/~blee/provdist/NobleGas/changelog_json.html#"
#test_alignment()


#print v2_row[0].value
#print indicator_map[v2_row[0].value]

#v1_workbook = xlrd.open_workbook(v1_file)
#v1_sheet = v1_workbook.sheet_by_index(0)
#v1_row = v1_sheet.row(4)

def write_modify(r1, r2, workbook, f_out, mode):
        if mode == 'r':
                out = u'''  <div about="Version2" rel="vo:hasAttribute">
    <div resource="v3:%s" typeof="vo:Attribute">
      <span style="font-weight:bold" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span>
      <table rel="vo:Undergoes">
'''%(r2[0].value, r2[0].value)
        elif mode == 'j':
                out = u'''
    <div about="v3:%s">
      <span style="font-weight:bold" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span>
      <table>
'''%(r2[0].value, r2[0].value)
        elif mode == 't':
                out = u"%s\n"%(r2[0].value)
        elif mode == 'u':
                out = u""

        if mode == 'r' or mode == 'j':
                out = out+'''        <tr>
```

```
          <th>Column v2</th>
          <th>Column v3</th>
          <th>Version 2</th>
          <th>Version 3</th>
        </tr>\n'''
        elif mode == 't':
                out = out+"Column v2\tColumn v3\tVersion 2\tVersion 3\n"
        f_out.write(out.encode('utf8'))
                #print '# Searching...'
                #print '# Comparing...'
        for i in range(0,54):
                if r2[i].value != r1[i].value:
                        #compare_print(j, v1_row[index_convert(j)].value, v2_row[j].value)
                        compare_print(mode, r2[0].value, r1[i].value, r2[i].value, workbook.split('/')[-1], i, i, f_out)
        if mode == 'r' or mode == 'j':
                f_out.write('  </table></div><br>\n')
        elif mode == 't' or mode == 'u':
                f_out.write("\n")

def write_removed(v2, col, row, f_out, mode):
        if mode == 'r' or mode == 'j':
                f_out.write('''
      <h3>Columns invalidated by %s</h3>
      <table about="Version2">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nColumns invalidated by %s\n"%(v2.split('/')[-1]))

        print "Removed Column"
        for i in col:
                v1_value = labels.get(i, "")
                if mode == 'r':
```

```
                                out = u'''        <tr resource="InvlidateChange%i" rev="vo:invalidatedBy" typeof="vo:InvalidateChange">
                <td resource="Attribute%i" rev="vo:Undergoes" typeof="vo:Attribute">%i</td>
          <td about="Attribute%i" property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
          <span about="Version1" property="vo:hasAttribute" resource="Attribute%i"/>
        </tr>
'''%(i, i, i, i, v1_value, i)
                        elif mode == 'j':
                                out = u'''        <tr id="InvlidateChange%i" about="InvlidateChange%i">
          <td>%i</td>
          <td>%s</td>
          <script type="application/ld+json">
'''%(i, i, i, v1_value)
                        elif mode == 't':
                                out = u"%i\t%s\n"%(i, v1_value)
                        elif mode == 'u':
                                out = u"""<http://example.com/NG/Version2> vo:hasAttribute <http://example.com/NG/Version2/Column%i> .
<http://example.com/NG/Version2/Column%i> vo:undergoes <http://example.com/Changelog#InvalidateChange%i> .
<http://example.com/Changelog#InvalidateChange%i> a vo:InvalidateChange ;
        vo:invalidatedBy <http://example.com/NG/Version3> .

"""%(i, i, i, i)
                        f_out.write(out.encode('utf8'))
                        if mode == 'j':
                                json1 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v2/Attribute", str(i)]) ,
```

```
"label": v1_value,
"undergoes":"".join([host, "InvalidateChange", str(i)]) ,
"@reverse" :    { "hasAttribute" : "Version2" }
}
                        json2 = {
"@context":context,
"@type":"vo:InvalidateChange" ,
"@id": "".join([host, "InvalidateChange", str(i)]) ,
"invalidatedBy"  :   "Version3"
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
''')


        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>
      <h3>Rows invalidated by %s</h3>
      <table about="Version2">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nRows invalidated by %s\n"%(v2.split('/')[-1]))
        elif mode == 'u':
                f_out.write("\n")

        print "Removed Row"
        for i, j in sorted(row):#i is row #, j is row id
                if mode == 'r':
                        out = u'''        <tr resource="InvlidateChange%s" rev="vo:invalidatedBy" typeof="vo:InvalidateChange">
```

```
          <td resource="Attribute%s" rev="vo:Undergoes" typeof="vo:Attribute">%i</td>
          <td about="Attribute%s" property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
          <span about="Version2" property="vo:hasAttribute" resource="Attribute%s"/>
        </tr>
'''%(j, j, i, j, j, j)
                elif mode == 'j':
                        out = u'''        <tr id="InvlidateChange%s" about="InvlidateChange%s">
          <td>%i</td>
          <td>%s</td>
          <script type="application/ld+json">
'''%(j, j, i,  j)
                elif mode == 't':
                        out = u"%i\t%s\n"%(i, j)
                elif mode == 'u':
                        out = u"""<http://example.com/NG/Version2> vo:hasAttribute <http://example.com/NG/Version2/%s> .
<http://example.com/NG/Version2/%s> vo:undergoes <http://example.com/Changelog#InvalidateChange%s> .
<http://example.com/Changelog#InvalidateChange%s> a vo:InvalidateChange ;
        vo:invalidatedBy <http://example.com/NG/Version2> .

"""%(j, j, j, j)
                f_out.write(out.encode('utf8'))
                if mode == 'j':
                        json1 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v2/Attribute", j]) ,
"label": j,
"undergoes":"".join([host, "InvalidateChange", j]) ,
"@reverse" :    { "hasAttribute" : "Version2" }
```

```
}
                        json2 = {
"@context":context,
"@type":"vo:InvalidateChange" ,
"@id": "".join([host, "InvalidateChange", j]) ,
"invalidatedBy"  :   "Version3"
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
''')
        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>

''')
        elif mode == 't' or mode == 'u':
                f_out.write("\n")


def write_added(v2, col, row, f_out, mode):
        if mode == 'r' or mode == 'j':
                f_out.write('''
      <h3>Columns added by %s</h3>
      <table about="Version2" rel="vo:absentFrom">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nColumns added by %s\n\n"%(v2.split('/')[-1]))

        print "Added Column"
        for i in col:
                print i#, v2_value
                if mode == 'r':
```

```
                        f_out.write('''        <tr about="AddChange%i" typeof="vo:AddChange">
          <td property="vo:resultsIn" resource="Attribute%i" typeof="vo:Attribute">%i</td>
          <td about="Attribute%i" property="http://www.w3.org/2000/01/rdf-schema#label"></td>
          <span about="Version3" property="vo:hasAttribute" resource="Attribute%i"/>
        </tr>
'''%(i, i, i, i, i))
                elif mode == 'j':
                        f_out.write('''        <tr id="AddChange%i" about="v2:Attribute%i">
          <td>%i</td>
          <td></td>
          <script type="application/ld+json">
'''%(i, i, i))
                        json1 = {
"@context":context,
"@type":"vo:AddChange" ,
"@id": "".join([host, "AddChange", str(i)]) ,
"resultsIn" :   "".join([ "http://ngdb.com/v3/Attribute", str(i)]),
"@reverse"  :   { "absentFrom": "Version2" }
}
                        json2 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v3/Attribute", str(i)]) ,
"label":"" ,
"@reverse" :    { "hasAttribute" : "Version3" }
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
```

```
''')
                elif mode == 't':
                        f_out.write("%i\t\n"%(i))
                elif mode == 'u':
                        f_out.write("""<http://example.com/NG/Version2> vo:absentFrom <http://example.com/Changelog#AddChange%i> .
<http://example.com/Changelog#AddChange%i> a vo:AddChange ;
        vo:resultsIn <http://example.com/NG/Version3/Column%s> .
<http://example.com/NG/Version3> vo:hasAttribute <http://example.com/NG/Version3/Column%s> .

"""%(i, i, i, i))
        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>
              <h3>Rows added by %s</h3>
              <table about="Version2" rel="vo:absentFrom">
'''%(v2.split('/')[-1]))
        elif mode == 't':
                f_out.write("\nRows added by %s\n\n"%(v2.split('/')[-1]))
        elif mode == 'u':
                f_out.write("\n")

        print "Added Row"
        for i, j in row:#i is the row #, j is the id
                if mode == 'r':                        #print i, v2_sheet.cell(i,0).value
                        out = u'''        <tr about="AddChange%s" typeof="vo:AddChange">
          <td property="vo:resultsIn" resource="Attribute%s" typeof="vo:Attribute">%i</td>
          <td about="Attribute%s" property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
          <span about="Version3" property="vo:hasAttribute" resource="Attribute%s"/>
```

```
        </tr>
'''%(j, j, i, j, j, j)
                elif mode == 'j':
                        out = u'''        <tr id="AddChange%s" about="v3:Attribute%s">
          <td>%i</td>
          <td property="http://www.w3.org/2000/01/rdf-schema#label">%s</td>
          <script type="application/ld+json">
'''%(j, j, i, j)
                elif mode == 't':
                        out = u"%i\t%s\n"%(i, j)
                elif mode == 'u':
                        out = u"""<http://example.com/NG/Version2> vo:absentFrom <http://example.com/Changelog#AddChange%s> .
<http://example.com/Changelog#AddChange%s> a vo:AddChange ;
        vo:resultsIn <http://example.com/NG/Version3/%s> .
<http://example.com/NG/Version3> vo:hasAttribute <http://example.com/NG/Version3/%s> .

"""%(j, j, j, j)
                f_out.write(out.encode('utf8'))
                if mode == 'j':
                        json1 = {
"@context":context,
"@type":"vo:AddChange" ,
"@id": "".join([host, "AddChange", j]) ,
"resultsIn" :   "".join([ "http://ngdb.com/v3/Attribute", j]),
"@reverse"  :   { "absentFrom": "Version2" }
}
                        json2 = {
"@context":context,
"@type":"vo:Attribute" ,
"@id":"".join(["http://ngdb.com/v3/Attribute", j]) ,
"label": j ,
"@reverse" :    { "hasAttribute" : "Version2" }
```

```
}
                        json.dump([json1, json2], f_out, indent=4, sort_keys=True)
                        f_out.write('''
          </script>
        </tr>
''')

        if mode == 'r' or mode == 'j':
                f_out.write('''      </table>
''')
        elif mode == 't' or mode == 'u':
                f_out.write("\n")

def write_header(f_out, mode):
        if mode == 'j' or mode == 'r':
                f_out.write('''<html>
  <head>
  </head>
  <body vocab="http://www.w3.org/nw/prov#" prefix="vo: https://orion.tw.rpi.edu/~blee/VersionOntology.owl# v2: http://ngdb.com/v2/ v3: http://ngdb.com/v3/">
''')
        if mode == 'j':
                f_out.write('''  <script type="application/ld+json">
''')
                json1 = {
"@context":context,
"@type":"vo:Version",
"@id":"Version2",
"label":"DB_final-55-7262_2015_03_08.xlsx"
}
                json2 = {
"@context":context,
"@type":"vo:Version",
"@id":"Version3",
```

```
"label":"NG_DB_final_2017_07_01.xlsx"
}
                json.dump([json1,json2], f_out, indent=4, sort_keys=True)
                f_out.write("\n  </script>\n")
        if mode == 'u':
                f_out.write("""@prefix vo: <http://orion.tw.rpi.edu/~blee/VersionOntology.owl#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.com/NG/Version3> a vo:Version ;
        skos:prefLabel "NG_DB_final_2017_07_01.xlsx" .

<http://example.com/NG/Version2> a vo:Version ;
        skos:prefLabel "DB_final-55-7262_2015_03_08.xlsx" .

""")

def write_footer(f_out, mode):
        if mode == 'r':
                f_out.write('</body>\n</html>')

def get_indicator_map(excel_files):
        indicator_map = {}
        for excel_file in excel_files:
                print 'Importing: ' + excel_file
                file_workbook = xlrd.open_workbook(excel_file)
                file_sheet = file_workbook.sheet_by_index(0)
                indicators = file_sheet.col(0)
                for i in range(4, file_sheet.nrows):
                        indicator_map[indicators[i].value] = excel_file
```

```
        return indicator_map

def compare(v1s, v2, fn_out, mode):
        v1_workbook = xlrd.open_workbook(v1s)
        v1_sheet = v1_workbook.sheet_by_index(0)
        i_keys = {j.value:i for i,j in enumerate(v1_sheet.col(0)[3:],3)}

        v2_workbook = xlrd.open_workbook(v2)
        v2_sheet = v2_workbook.sheet_by_index(0)
        v2_keys = {j.value:i for i,j in enumerate(v2_sheet.col(0)[3:],3)}

        f_out = open(fn_out, 'w')

        new_col = [i for i in range(0, v2_sheet.ncols) if index_convert(i) == -1]
        new_row = [(v2_keys[i], i) for i in v2_keys.keys() if i not in i_keys.keys()]
        old_col = [i for i in range(0, v1_sheet.ncols) if index_convert(i) == -1]
        old_row = [(i_keys[i], i) for i in i_keys.keys() if i not in v2_keys.keys()]

        write_header(f_out, mode)
        write_added(v2, new_col, new_row, f_out, mode)
        write_removed(v2, old_col, old_row, f_out, mode)

        if mode == 'r' or mode == 'j':
                f_out.write('''
      <h3>Change Log</h3>
''')
        elif mode == 't':
                f_out.write("Change Log\n")

        workbook_name = ''
```

```
        for i in range(3,v2_sheet.nrows):
                v2_row = v2_sheet.row(i)
                #workbook_name = v1_file
                if v2_row[0].value in [j for i, j in new_row] or v2_row[0].value in [j for i, j in old_row]:
                        continue
                v1_row = v1_sheet.row(i_keys[v2_row[0].value])
                write_modify(v1_row, v2_row, workbook_name, f_out, mode)

        write_footer(f_out, mode)
        f_out.close()

if __name__ == "__main__":
        if '-json' in sys.argv:
                mode = 'j'
                out_name = 'isotope2_3_json.html'
        elif '-rdfa' in sys.argv:
                mode = 'r'
                out_name = 'isotope2_3_rdfa.html'
        elif '-txt' in sys.argv:
                mode = 't'
                out_name = 'changelog2_3.txt'
        elif '-ttl' in sys.argv:
                mode = 'u'
                out_name = 'changelog2_3.ttl'

        v2_dir = join(separator, 'data', 'NGdata', 'v2')
        v3_dir = join(separator, 'data', 'NGdata', 'v3')

        v2_file = join(v2_dir, 'DB_final-55-7262_2015_03_08.xlsx')
        v3_file = join(v3_dir, 'NG_DB_final_2017_07_01.xlsx')

        compare(v2_file, v3_file, out_name, mode)
```

APPENDIX C
GLOBAL CHANGE MASTER DIRECTORY CHANGE LOG GENERATOR VERSION JUNE 12, 2012 TO VERSION 8.4.1

```
import glob, json, rdflib, re
from rdflib import URIRef, Literal, Namespace
from rdflib.namespace import RDF, SKOS

def GCMDChangeLogGenerator(GCMDfile):
        GCMD = Namespace("http://gcmdservices.gsfc.nasa.gov/kms/concept/")

        #GCMDfile = ['GCMD8_3.rdf', 'GCMD8_4.rdf','GCMD8_4_1.rdf']
        numbers = [re.search('GCMD(.*).rdf', i).group(1).replace("_","") for i in GCMDfile]
        print numbers
        filename = 'ChangelogGCMD'+"_".join(numbers)+'.html'
        output = open(filename, 'w') #'/home/blee/provdist/GCMD/webGCMD83_84.html', 'w')
        #output = codecs.open('/home/blee/GCMD/GCMD8_3to8_4.html', mode='w', encoding='utf-8')

        g0 = rdflib.Graph()
        g0.parse(GCMDfile[0])
        g1 = rdflib.Graph()
        g1.parse(GCMDfile[1])

        ver = [re.search('GCMD(.*).rdf', i).group(1).replace("_",".") for i in GCMDfile]#['8.3', '8.4']
        print ver
        new = g1-g0
        old = g0-g1
```

```

        #Get the date for the change notes in the new changes made in version 2
        #This will help determine if some concepts were changed without being moved
        #Their change notes should have a date on the same day as the new additions.
        #This is probably a bad way of determining this.
        date = g1.value(new.value(predicate=RDF.type, object=SKOS.Concept), SKOS.changeNote).split()[0]
        context = "https://orion.tw.rpi.edu/~blee/provdist/GCMD/VO.jsonld"

        #################
        ###   Header   ###
        #################

        output.write('''<html>
          <head>
            <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
          </head>
          <body vocab="http://www.w3.org/nw/prov#" prefix="gcmd: http://gcmdservices.gsfc.nasa.gov/kms/concept/">
            <h2 property="http://purl.org/dc/terms/title">
              <span about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span> to
              <span about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span>
        '''%(ver[0], GCMDfile[0], ver[1], GCMDfile[1]))
```

```

        output.write('''      <script type="application/ld+json">
        [
                {
                        "@context" : "%s" ,
                        "type"          :         "vo:Version" ,
                        "id"         :        "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" ,
                        "label" :       "%s"
                },
                {
                        "@context" : "%s",
                        "type"   :       "vo:Version" ,
                        "id"    :   "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" ,
                        "label" :       "%s"
                }
        ]
              </script>
            </h2>
        '''%(context, ver[0], GCMDfile[0], context, ver[1], GCMDfile[1]))

        ################
        ###   ADDED   ###
        ################

        output.write('''
              <h3>Concepts added to %s</h3>
              <table about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" class="table table-striped">
                <tr>
                  <th>Link</th>
                  <th>Concept</th>
```

```
                    <th>Change Note</th>
                  </tr>
        '''%(GCMDfile[1], ver[1]))

        c = 0

        for i in new.subjects(RDF.type, SKOS.Concept):
                changeNote = "<br>\n".join(g1.objects(i, SKOS.changeNote))
                output.write((u'''        <tr id="AddChange%i" about="%s?version=%s">
                    <td>
                      <a href="%s?version=%s">Link</a>
                    </td>
                    <td property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
                    <td property="http://www.w3.org/2004/02/skos/core#changeNote">%s</td>
        '''%(c, str(i), ver[1], str(i), ver[1], g1.value(i, SKOS.prefLabel), changeNote)).encode('utf8'))
                output.write((u'''
                    <script type="application/ld+json">
        [
                {
                        "@context" : "%s" ,
                        "type"       :       "vo:AddChange" ,
                        "id"       :       "this:AddChange%i" ,
                        "resultsIn" :       "gcmd:%s?version=%s" ,
                        "@reverse"  :       { "absentFrom": "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                },
                {
                        "@context" : "%s" ,
                        "type"       :       "vo:Attribute" ,
                        "id"       :       "gcmd:%s?version=%s" ,
                        "label" :       "%s" ,
```

```
                        "@reverse" :        { "hasAttribute" : "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                }
        ]
                    </script>
                  </tr>
        '''%(context, c, i.split('/')[-1], ver[1], ver[0], context, i.split('/')[-1], ver[1], g1.value(i, SKOS.prefLabel), ver[1])).encode('utf8'))
                c += 1

        output.write('''      </table>

        ''')

        #print date

        #################
        ### REMOVED   ###
        #################

        output.write('''
              <h3>Concepts removed from %s</h3>
              <table about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" class="table table-striped">
                <tr>
                  <th>Link</th>
                  <th>Concept</th>
                </tr>
        '''%(GCMDfile[0], ver[0]))

        c = 0

        for i in old.subjects(RDF.type, SKOS.Concept):#Reverse relations due to ordering and structure
```

```
                output.write((u'''        <tr id="InvlidateChange%i" about="%s?version=%s">
                  <td>
                    <a href="%s?version=%s">Link</a>
                  </td>
                  <td property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
        '''%(c, str(i), ver[0], str(i), ver[0], g0.value(i, SKOS.prefLabel), )).encode('utf8'))
                output.write((u'''          <script type="application/ld+json">
        [
                {
                        "@context" : "%s" ,
                        "type"         :        "vo:Attribute" ,
                        "id"         :        "gcmd:%s?version=%s" ,
                        "label"         :        "%s" ,
                        "undergoes" :        "this:InvalidateChange%i" ,
                        "@reverse" :        { "hasAttribute" : "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                },
                {
                        "@context" : "%s" ,
                        "type"         :        "vo:InvalidateChange" ,
                        "id"         :        "this:InvalidateChange%i" ,
                        "invalidatedBy"         :        "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s"
                }
        ]
                  </script>
                </tr>
```

```
        '''%(context, i.split('/')[-1], ver[0], g0.value(i,
↪   SKOS.prefLabel), c, ver[0], context, c, ver[1])).encode('utf8'))
                c += 1
        output.write('''      </table>

        ''')

        ##################
        ###   Modify   ###
        ##################

        output.write('''
              <h3>Moved Concepts</h3>
              <table class="table table-striped">
                <tr>
                  <th>Link v1</th>
                  <th>Link v2</th>
                  <th>Label</th>
                  <th>Old Parent</th>
                  <th>New Parent</th>
                </tr>\n
        ''')

        c = 0

        for i in g1.subjects(RDF.type, SKOS.Concept):
                if (i, None, None) in g0:
                        b0 = g0.value(i, SKOS.broader)
                        b1 = g1.value(i, SKOS.broader)
                        if b0 != b1:
                                output.write((u'''        <tr
↪   id="MoveChange%i" about="%s?version=%s">
                  <td><a href=%s?version=%s>Link</a></td>
                  <td><a href=%s?version=%s>Link</a></td>
                  <td
↪   property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
```

```
                    <td about="%s?version=%s" property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
                    <td about="%s?version=%s" property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
        '''%(c, str(i), ver[1],
             str(i), ver[0],
             str(i), ver[1],
             g1.value(i, SKOS.prefLabel),
             b0, ver[0], g0.value(b0, SKOS.prefLabel),
             b1, ver[1], g1.value(b1, SKOS.prefLabel))).encode('utf8'))

                            output.write((u'''          <script type="application/ld+json">
        [
                {
                        "@context" : "%s" ,
                        "type"         :         "vo:Attribute" ,
                        "id"         :         "gcmd:%s?version=%s" ,
                        "label"         :         "%s" ,
                        "undergoes" :         "this:MoveChange%i" ,
                        "@reverse" :         { "hasAttribute" : "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                },
                {
                        "@context" : "%s" ,
                        "type"         :         "vo:MoveChange" ,
                        "id"         :         "this:MoveChange%i" ,
                        "resultsIn"         :         "gcmd:%s?version=%s"
                },
                {
                        "@context" : "%s" ,
                        "type"         :         "vo:Attribute" ,
                        "id"         :         "gcmd:%s?version=%s" ,
                        "label"         :         "%s" ,
```

```
                    "@reverse" :        { "hasAttribute" : "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                }
        ]
                  </script>
                </tr>
        '''%(context, i.split('/')[-1], ver[0], g0.value(i, SKOS.prefLabel), c, ver[0],
             context, c, i.split('/')[-1], ver[1],
             context, i.split('/')[-1], ver[1], g1.value(i, SKOS.prefLabel), ver[1])  ).encode('utf8'))
                        c += 1

        output.write('''      </table>

        ''')

        ##################################
        ###   NON-STRUCTURAL CHANGES   ###
        ##################################

        output.write('''
              <h3>Non-Structural Changes</h3>
              <table class="table table-striped">
                <tr>
                  <th>Link v1</th>
                  <th>Link v2</th>
                  <th>Label</th>
                  <th>Change Notes</th>
                </tr>\n
        ''')

        c = 0

        for i in g1.subjects(RDF.type, SKOS.Concept):
                if (i, None, None) in g0:
```

```
                    b0 = g0.value(i, SKOS.broader)
                    b1 = g1.value(i, SKOS.broader)
                    if b0 == b1:
                            new_note = False
                            notes = []
                            for note in g1.objects(i, SKOS.changeNote):
                                    note_date = note.split()[0]
                                    #print note_date
                                    if note_date == date:
                                            new_note = True
                                            notes.append(note)
                            if new_note:
                                    output.write((u''' <tr id="ModifyChange%i" about="%s?version=%s">
              <td><a href=%s?version=%s>Link</a></td>
              <td><a href=%s?version=%s>Link</a></td>
              <td property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
              <td property="http://www.w3.org/2004/02/skos/core#changeNote">%s</td>
    '''%(c, str(i), ver[1],
         str(i), ver[0],
         str(i), ver[1],
         g1.value(i, SKOS.prefLabel),
         "<br>\n              ".join(notes)
    )).encode('utf8'))
                                    output.write((u'''
            </tr>
    '''%()).encode('utf8'))
                                    c += 1

    output.write('''      </table>

    ''')
    output.write("\t</body>\n</html>")
```

```
        output.close()

if __name__ == "__main__":
        GCMDfiles = sorted(glob.glob("*.rdf"))
        for i in range(len(GCMDfiles)-1):
                print "Starting",GCMDfiles[i-1],"and",GCMDfiles[i] #It's done this way because GCMDJun1220012 sorts to the last item
                GCMDChangeLogGenerator([GCMDfiles[i-1],GCMDfiles[i]])
```

APPENDIX D
GLOBAL CHANGE MASTER DIRECTORY CHANGE LOG GENERATOR VERSION 8.4.1 TO 8.5

```python
import glob, json, rdflib, re
from rdflib import URIRef, Literal, Namespace
from rdflib.namespace import RDF, SKOS

def GCMDChangeLogGenerator(GCMDfile):
        GCMD = Namespace("http://gcmdservices.gsfc.nasa.gov/kms/concept/")
        GCMD8_5 = Namespace("https://gcmdservices.gsfc.nasa.gov/kms/concept/")

        #GCMDfile = ['GCMD8_3.rdf', 'GCMD8_4.rdf','GCMD8_4_1.rdf']
        numbers = [re.search('GCMD(.*).rdf', i).group(1).replace("_","") for i in GCMDfile]
        print numbers
        filename = 'ChangelogGCMD2'+"_".join(numbers)+'.html'
        output = open(filename, 'w') #'/home/blee/provdist/GCMD/webGCMD83_84.html', 'w')
        #output = codecs.open('/home/blee/GCMD/GCMD8_3to8_4.html', mode='w', encoding='utf-8')

        g0 = rdflib.Graph()
        g0.parse(GCMDfile[0])
        g1 = rdflib.Graph()
        g1.parse(GCMDfile[1])

        ver = [re.search('GCMD(.*).rdf', i).group(1).replace("_",".") for i in GCMDfile]#['8.3', '8.4']
        print ver
        new = rdflib.Graph()
        for s, p, o in g1.triples((None, RDF.type, SKOS.Concept)):
```

```
            if not (GCMD[s.split('/')[-1]], p, o) in g0:
                    new.add((s, p, o))
    old = rdflib.Graph()
    for s, p, o in g0.triples((None, RDF.type, SKOS.Concept)):
            if not (GCMD8_5[s.split('/')[-1]], p, o) in g1:
                    old.add((s, p, o))

    #Get the date for the change notes in the new changes made in version 2
    #This will help determine if some concepts were changed without being moved
    #Their change notes should have a date on the same day as the new additions.
    #This is probably a bad way of determining this.
    date = g1.value(new.value(predicate=RDF.type, object=SKOS.Concept), SKOS.changeNote).split()[0]
    context = "https://orion.tw.rpi.edu/~blee/provdist/GCMD/VO.jsonld"

    #################
    ###   Header   ###
    #################

    output.write('''<html>
      <head>
        <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
      </head>
      <body vocab="http://www.w3.org/nw/prov#" prefix="gcmd: http://gcmdservices.gsfc.nasa.gov/kms/concept/">
        <h2 property="http://purl.org/dc/terms/title">
```

```
            <span about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span> to
            <span about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" property="http://www.w3.org/2000/01/rdf-schema#label">%s</span>
        '''%(ver[0], GCMDfile[0], ver[1], GCMDfile[1]))

        output.write('''      <script type="application/ld+json">
        [
                {
                        "@context" : "%s" ,
                        "type"          :         "vo:Version" ,
                        "id"          :         "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" ,
                        "label" :        "%s"
                },
                {
                        "@context" : "%s",
                        "type"   :        "vo:Version" ,
                        "id"     :    "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" ,
                        "label" :        "%s"
                }
        ]
            </script>
          </h2>
        '''%(context, ver[0], GCMDfile[0], context, ver[1], GCMDfile[1]))

        #################
        ###   ADDED   ###
        #################

```

```
        output.write('''
              <h3>Concepts added to %s</h3>
              <table about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" class="table table-striped">
                <tr>
                  <th>Link</th>
                  <th>Concept</th>
                  <th>Change Note</th>
                </tr>
        '''%(GCMDfile[1], ver[1]))

        c = 0

        for i in new.subjects(RDF.type, SKOS.Concept):
                changeNote = "<br>\n".join(g1.objects(i, SKOS.changeNote))
                output.write((u'''        <tr id="AddChange%i" about="%s?version=%s">
                    <td>
                      <a href="%s?version=%s">Link</a>
                    </td>
                    <td property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
                    <td property="http://www.w3.org/2004/02/skos/core#changeNote">%s</td>
        '''%(c, str(i), ver[1], str(i), ver[1], g1.value(i, SKOS.prefLabel), changeNote)).encode('utf8'))
                output.write((u'''
                    <script type="application/ld+json">
        [
                {
                        "@context" : "%s" ,
                        "type"        :        "vo:AddChange" ,
                        "id"        :        "this:AddChange%i" ,
                        "resultsIn" :        "gcmd:%s?version=%s" ,
```

```
                    "@reverse"   :        { "absentFrom": "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                },
                {
                    "@context" : "%s" ,
                    "type"        :        "vo:Attribute" ,
                    "id"        :        "gcmd:%s?version=%s" ,
                    "label" :        "%s" ,
                    "@reverse" :        { "hasAttribute" : "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                }
        ]
                  </script>
                </tr>
        '''%(context, c, i.split('/')[-1], ver[1], ver[0], context, i.split('/')[-1], ver[1], g1.value(i, SKOS.prefLabel), ver[1])).encode('utf8'))
                c += 1

        output.write('''      </table>

        ''')

        #print date

        #################
        ### REMOVED   ###
        #################

        output.write('''
              <h3>Concepts removed from %s</h3>
              <table about="gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" class="table table-striped">
                <tr>
                  <th>Link</th>
```

```
                    <th>Concept</th>
                </tr>
        '''%(GCMDfile[0], ver[0]))

        c = 0

        for i in old.subjects(RDF.type, SKOS.Concept):#Reverse
↪   relations due to ordering and structure
                output.write((u'''        <tr id="InvlidateChange%i"
↪   about="%s?version=%s">
                    <td>
                      <a href="%s?version=%s">Link</a>
                    </td>
                    <td
↪   property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
        '''%(c, str(i), ver[0], str(i), ver[0], g0.value(i,
↪   SKOS.prefLabel), )).encode('utf8'))
                output.write((u'''          <script
↪   type="application/ld+json">
        [
                {
                        "@context" : "%s" ,
                        "type"        :        "vo:Attribute" ,
                        "id"        :        "gcmd:%s?version=%s" ,
                        "label"        :        "%s" ,
                        "undergoes"
↪   :        "this:InvalidateChange%i" ,
                        "@reverse" :        { "hasAttribute" :
↪   "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                },
                {
                        "@context" : "%s" ,
                        "type"        :        "vo:InvalidateChange"
↪   ,
```

```
                        "id"        :        "this:InvalidateChange%i",
                        "invalidatedBy"        :        "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s"
                    }
            ]
                      </script>
                    </tr>
        '''%(context, i.split('/')[-1], ver[0], g0.value(i, SKOS.prefLabel), c, ver[0], context, c, ver[1])).encode('utf8'))
                    c += 1
        output.write('''      </table>

        ''')

        #################
        ###   Modify   ###
        #################

        output.write('''
                  <h3>Moved Concepts</h3>
                  <table class="table table-striped">
                    <tr>
                      <th>Link v1</th>
                      <th>Link v2</th>
                      <th>Label</th>
                      <th>Old Parent</th>
                      <th>New Parent</th>
                    </tr>\n
        ''')

        c = 0

        for i in g1.subjects(RDF.type, SKOS.Concept):
                    i_ = GCMD[i.split('/')[-1]]
```

```
            if (i_, None, None) in g0:
                    b0 = g0.value(i_, SKOS.broader)
                    b1 = g1.value(i, SKOS.broader)
                    if b1 != None:
                            b1_ = GCMD[b1.split('/')[-1]]
                    if b0 != b1_:
                            output.write((u'''        <tr id="MoveChange%i" about="%s?version=%s">
              <td><a href=%s?version=%s>Link</a></td>
              <td><a href=%s?version=%s>Link</a></td>
              <td property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
              <td about="%s?version=%s" property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
              <td about="%s?version=%s" property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
    '''%(c, str(i), ver[1],
         str(i), ver[0],
         str(i), ver[1],
         g1.value(i, SKOS.prefLabel),
         b0, ver[0], g0.value(b0, SKOS.prefLabel),
         b1, ver[1], g1.value(b1, SKOS.prefLabel)) ).encode('utf8'))

                            output.write((u'''          <script type="application/ld+json">
    [
            {
                    "@context" : "%s" ,
                    "type"        :        "vo:Attribute" ,
                    "id"        :        "gcmd:%s?version=%s" ,
                    "label"        :        "%s" ,
                    "undergoes" :        "this:MoveChange%i" ,
                    "@reverse" :        { "hasAttribute" : "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
            },
```

```
                {
                        "@context" : "%s" ,
                        "type"        :        "vo:MoveChange" ,
                        "id"        :        "this:MoveChange%i" ,
                        "resultsIn"        :        "gcmd:%s?version=%s"
                },
                {
                        "@context" : "%s" ,
                        "type"        :        "vo:Attribute" ,
                        "id"        :        "gcmd:%s?version=%s" ,
                        "label"        :        "%s" ,
                        "@reverse" :        { "hasAttribute" : "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
                }
        ]
                  </script>
                </tr>
        '''%(context, i.split('/')[-1], ver[0], g0.value(i_, SKOS.prefLabel), c, ver[0],
                context, c, i.split('/')[-1], ver[1],
                context, i.split('/')[-1], ver[1], g1.value(i, SKOS.prefLabel), ver[1])  ).encode('utf8'))
                                c += 1

        output.write('''      </table>

        ''')

        output.write('''
              <h3>Modified Concepts</h3>
              <table class="table table-striped">
                <tr>
                  <th>Link v1</th>
                  <th>Link v2</th>
                  <th>Label</th>
```

```
                </tr>\n
        ''')

        c = 0


        ##################################
        ###   NON-STRUCTURAL CHANGES   ###
        ##################################

        output.write('''
              <h3>Non-Structural Changes</h3>
              <table class="table table-striped">
                <tr>
                  <th>Link v1</th>
                  <th>Link v2</th>
                  <th>Label</th>
                  <th>Change Notes</th>
                </tr>\n
        ''')

        c = 0

        for i in g1.subjects(RDF.type, SKOS.Concept):
                i_ = GCMD[i.split('/')[-1]]
                if (i_, None, None) in g0:
                        b0 = g0.value(i_, SKOS.broader)
                        b1 = g1.value(i, SKOS.broader)
                        if b1 != None:
                                b1_ = GCMD[b1.split('/')[-1]]
                        if b0 == b1_ and i != i_:
                                output.write((u'''        <tr id="NameChange%i" about="%s?version=%s">
                  <td><a href=%s?version=%s>Link</a></td>
                  <td><a href=%s?version=%s>Link</a></td>
```

```
                <td
↪ property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
    '''%(c, str(i), ver[1],
         str(i_), ver[0],
         str(i), ver[1],
         g1.value(i, SKOS.prefLabel)
            )))
                            output.write((u'''          <script
↪ type="application/ld+json">
    [
            {
                    "@context" : "%s" ,
                    "type"          :       "vo:Attribute" ,
                    "id"          :       "%s?version=%s" ,
                    "label"          :       "%s" ,
                    "undergoes" :       "this:NameChange%i" ,
                    "@reverse" :       { "hasAttribute" :
↪ "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
            },
            {
                    "@context" : "%s" ,
                    "type"          :       "vo:ModifyChange" ,
                    "id"          :       "this:NameChange%i" ,
                    "resultsIn"          :       "%s?version=%s"
            },
            {
                    "@context" : "%s" ,
                    "type"          :       "vo:Attribute" ,
                    "id"          :       "%s?version=%s" ,
                    "label"          :       "%s" ,
                    "@reverse" :       { "hasAttribute" :
↪ "gcmd:concept_scheme/sciencekeywords/?format=xml&version=%s" }
            }
    ]
                </script>
            </tr>
```

```
    '''%(context, i_, ver[0], g0.value(i_, SKOS.prefLabel), c, ver[0],
         context, c, i, ver[1],
         context, i, ver[1], g1.value(i, SKOS.prefLabel), ver[1])  ).encode('utf8'))
                            c += 1

    output.write('''      </table>

    ''')

    for i in g1.subjects(RDF.type, SKOS.Concept):
            i_ = GCMD[i.split('/')[-1]]
            if (i_, None, None) in g0:
                    b0 = g0.value(i_, SKOS.broader)
                    b1 = g1.value(i, SKOS.broader)
                    if b1 != None:
                            b1_ = GCMD[b1.split('/')[-1]]
                    if b0 == b1_:
                            new_note = False
                            notes = []
                            for note in g1.objects(i, SKOS.changeNote):
                                    note_date = note.split()[0]
                                    #print note_date
                                    if note_date == date:
                                            new_note = True
                                            notes.append(note)
                            if new_note:
                                    output.write((u'''      <tr id="ModifyChange%i" about="%s?version=%s">
              <td><a href=%s?version=%s>Link</a></td>
              <td><a href=%s?version=%s>Link</a></td>
              <td property="http://www.w3.org/2004/02/skos/core#prefLabel">%s</td>
```

```
                    <td property="http://www.w3.org/2004/02/skos/core#changeNote">%s</td>
        '''%(c, str(i), ver[1],
             str(i), ver[0],
             str(i), ver[1],
             g1.value(i, SKOS.prefLabel),
             "<br>\n              ".join(notes)
        )).encode('utf8'))
                                        output.write((u'''
                  </tr>
        '''%()).encode('utf8'))
                                        c += 1

        output.write('''      </table>

        ''')
        output.write("\t</body>\n</html>")
        output.close()

if __name__ == "__main__":
        GCMDfiles = sorted(glob.glob("*.rdf"))
        GCMDfiles = ["GCMD8_5.rdf", "GCMD8_4_1.rdf"]
        for i in range(len(GCMDfiles)-1):
                print "Starting",GCMDfiles[i-1],"and",GCMDfiles[i] #It's done this way because GCMDJun1220012 sorts to the last item
                GCMDChangeLogGenerator([GCMDfiles[i-1],GCMDfiles[i]])
```

APPENDIX E
TURTLE EXTRACTOR

```
from bs4 import BeautifulSoup
import glob, rdflib, json, re

def extracting(f, d):
        #f = 'ChangelogGCMD70_80.html'
        #d = 'Graph'+re.search('ChangelogGCMD(.*).html',
    ↪ f).group(1)+'.ttl'

        fp = open(f)
        soup = BeautifulSoup(fp, 'html5lib')
        fp.close()

        print 'extracting...'
        js = soup.find_all('script')
        items = [item for sublist in js for item in
    ↪ json.loads(sublist.text)]

        print 'loading...'
        g = rdflib.Graph()
        g.parse(data = json.dumps(items), format='json-ld')

        print 'writing...'
        g.serialize(destination=d, format='turtle')
        print 'written'

if __name__ == "__main__":
        l = glob.glob('Changelog*.html')
        for i in l:
                d = 'Graph'+re.search('ChangelogGCMD(.*).html',
    ↪ i).group(1)+'.ttl'
                print "Extracting: "+i
```

```
                    print "\tto", d
                    extracting(i, d)
```

APPENDIX F
MARINE BIODIVERSITY VIRTUAL LABORATORY CLASSIFIER COMPARISON

```
import urllib

class entry:
	query = None
	dist = None
	freq = None
	tax = None
	poss = None

	def __lt__(self, other):
		if self.query == other.query:
			return self.freq < other.freq
		else:
			return self.query < other.query

def is_number(s):
	try:
		float(s)
		return True
	except ValueError:
		return False

def file_parse(f_name):
	f = open(f_name, 'r')
	found = {}
	for i in mock:
		found[i] = [[],[],0]
	ambiguous = []
	fp = []
	fn_count = 0
```

```
        fn = []
        results = {}
        total = 0
        if f_name.split('_')[1] == 'spingo':
            if f_name == "silva_spingo":
                family[4] = 'Clostridiaceae_1'
            else:
                family[2] = 'Bacillaceae 1'
                family[4] = 'Clostridiaceae 1'
            for line in f:
                x = line.split('\t')
                entry_id = x[0].split('|')[0]
                results[entry_id] = entry()
                results[entry_id].query = x[0]
                results[entry_id].freq = int(results[entry_id].query.split('|')[-1].split(':')[1])
                results[entry_id].dist = float(x[1])
                results[entry_id].tax = x[2:]

                total += results[entry_id].freq

                if not is_number(results[entry_id].tax[-1]):
                    results[entry_id].poss = results[entry_id].tax[-1].split(',')
                    results[entry_id].tax = results[entry_id].tax[:-1]
                    if results[entry_id].tax[-2] != 'AMBIGUOUS':
                        print results[entry_id].tax[-2]

                clean_tax = [x for x in results[entry_id].tax if not is_number(x)]
                if len(clean_tax) < 7:
                    clean_tax[0:0] = ['Pass', 'Pass', 'Pass', 'Pass']
```

```
                del clean_tax[5]
                results[entry_id].tax = clean_tax

                if clean_tax[4] == 'AMBIGUOUS':
                    fn_count += 1
                    ambiguous.append(entry_id)
                elif not clean_tax[4] in family:
                    fp.append((entry_id, 'family'))
                elif clean_tax[5] == 'AMBIGUOUS':
                    for i in range(len(family)):
                        if clean_tax[4] == family[i]:
                            found[mock[i]][0].append(entry_id)
                elif not clean_tax[5] in genus:
                    fp.append((entry_id, 'genus'))
                elif clean_tax[6] == 'AMBIGUOUS':
                    for i in range(len(genus)):
                        if clean_tax[5] == genus[i]:
                            found[mock[i]][1].append(entry_id)
                elif not ' '.join(clean_tax[6].split('_')) in mock:
                    fp.append((entry_id, 'species'))
                else:
                    found[' '.join(clean_tax[6].split('_'))][2] += 1

        elif f_name.split('_')[1] == 'gast':
            skip = False
            if f_name == "rdp_gast":
                family[2] = 'Bacillaceae 1'
                family[4] = 'Clostridiaceae 1'
            for line in f:
                if not skip:
                    skip = True
                    continue
```

```
x = line.split('\t')
entry_id = x[0].split('|')[0]
results[entry_id] = entry()
results[entry_id].query = x[0]
results[entry_id].freq = int(results[entry_id].query.split('|')[-1].split(':')[1])
results[entry_id].dist = float(x[2])
results[entry_id].tax = x[1].split(';')

total += results[entry_id].freq
temp_tax = results[entry_id].tax

if len(temp_tax) < 5:
    fn_count += 1
    ambiguous.append(entry_id)
elif not temp_tax[4] in family:
    fp.append((entry_id, 'family'))
elif len(temp_tax) < 6:
    for i in range(len(family)):
        if temp_tax[4] == family[i]:
            found[mock[i]][0].append(entry_id)
elif not temp_tax[5] in genus:
    fp.append((entry_id, 'genus'))
elif len(temp_tax) < 7:
    for i in range(len(genus)):
        if temp_tax[5] == genus[i]:
            found[mock[i]][1].append(entry_id)
elif not '
'.join(results[entry_id].tax[5:7]) in mock:
    fp.append((entry_id, 'species'))
else:
    found['
'.join(results[entry_id].tax[5:7])][2] += 1

```

```
        fp_file = open("fp_"+f_name+".txt", 'w')
        for fid, frank in fp:
                fp_file.write("\t".join([fid, str(results[fid].freq), frank, str(results[fid].tax)])+"\n")
        fp_file.close()

        for i in found.keys():
                if sum([len(found[i][0]), len(found[i][1]), found[i][2]]) == 0:
                        fn.append(i)
        print fn
        ttl_output(f_name, fp, fn, found, results)
        total = len(results.keys())
        print "\t".join(["False Positives:", str(len(fp))])
        print "\t".join(["Ambiguous:", str(len(ambiguous))])
        print "\t".join(["Total:", str(len(results.keys()))])
        coverage = sum([len(found[i][0])+len(found[i][1])+found[i][2] for i in found.keys()])
        print "\t".join(["Coverage:", str(coverage)])
        print "\t".join(["Percentage:", str(float(coverage)/total*100)])
        for k in sorted(mock):
                print "\t".join([k+":", str(len(found[k][0])), str(len(found[k][1])), str(found[k][2])])

        return results

def ttl_output(f_name, false_positive, false_negative, found, results):
        fp_file = open("fp_"+f_name+".ttl", 'w')
        fp_file.write("""@prefix vo: <http://orion.tw.rpi.edu/~blee/VersionOntology.owl#> .
@prefix mbvl: <http://example.com/MBVL/> .
@prefix next: <http://example.com/MBVL/%s> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
```

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.com/MBVL/db> a vo:Version .
<http://example.com/MBVL/%s> a vo:Version .\n"""%(f_name, f_name))
        for i in range(len(false_positive)):
                fid, j = false_positive[i]
                clean_fid = urllib.quote(fid.split()[0])
                if results[fid].freq > 10:
                        fp_file.write("""mbvl:db vo:absentFrom mbvl:AddChange%i .
next:%s a vo:Attribute .
mbvl:%s vo:hasAttribute next:%s .
mbvl:AddChange%i a vo:AddChange ;
        vo:resultsIn next:%s .\n
"""%(i, clean_fid, f_name, clean_fid, i, clean_fid))
        fp_file.close()

def get_output(tax1, alg1, tax2, alg2):
        if alg1 == 'silva_spingo':
                t1 = [x for x in tax1 if not is_number(x)]
                del t1[5]
        elif alg1 == 'rdp_spingo':
                t1 = [x for x in tax1 if not is_number(x)]
                print t1
                del t1[1]
                t1[0:0] = ['Pass', 'Pass', 'Pass', 'Pass']
        else:
                t1 = tax1

        if alg2 == 'silva_spingo':
                t2 = [x for x in tax2 if not is_number(x)]
                del t2[5]
        elif alg2 == 'rdp_spingo':
```

```
            t2 = [x for x in tax2 if not is_number(x)]
            del t2[1]
            t2[0:0] = ['Pass', 'Pass', 'Pass', 'Pass']
    else:
            t2 = tax2

    if alg1.split('_')[1] != alg2.split('_')[1]:
            for j in range(7):
                    if t1[j] == 'Pass':
                            continue
                    if j >= len(t2):
                            if t1[j] != 'AMBIGUOUS':
                                    return "\t".join([ '---', rank[j], str(t1[j:])])
                    elif j == 6:
                            if t1[j] == 'AMBIGUOUS':
                                    return "\t".join(['+++', rank[j], str(t2[j:])])
                            elif t1[j] != '_'.join(t2[5:7]):
                                    return "\t".join([ '>>>', rank[j], t1[j], str(tax2[5:])])
                    elif t1[j] == 'AMBIGUOUS':
                            return "\t".join([ '+++', rank[j], str(t2[j:])])
                    elif t1[j] != t2[j]:
                            return "\t".join([ '>>>', rank[j], str(t1[j]), str(t2[j:])])
            return None
    elif alg1.split('_')[1] == 'spingo':
            for j in range(7):
                    if t1[j] == 'Pass' or t2[j] == 'Pass':
                            continue
                    if t1[j] == 'AMBIGUOUS':
                            if t2[j] == 'AMBIGUOUS':
                                    return None
                            else:
```

```
                    return "\t".join(['+++', rank[j], str(t2[j:])])
            elif t2[j] == 'AMBIGUOUS':
                return "\t".join(['---', rank[j], str(t1[j:])])
            elif t1[j] != t2[j]:
                return "\t".join(['>>>', rank[j], str(t1[j]), str(t2[j])])
        return None
    elif alg1.split('_')[1] == 'gast':
        if len(t1) > len(t2):
            return "\t".join(['---', rank[len(t2)], str(t1[len(t2):])])
        elif len(t1) < len(t2):
            return "\t".join(['+++', rank[len(t1)], str(t2[len(t1):])])
        else:
            for j in range(len(t1)):
                if t1[j] != t2[j]:
                    return "\t".join(['>>>', rank[j], str(t1[j]), str(t2[j])])
        return None
    else:
        return -1

mock = ['Acinetobacter baumannii',
        'Actinomyces odontolyticus',
        'Bacillus cereus',
        'Bacteroides vulgatus',
        'Clostridium beijerinckii',
        'Deinococcus radiodurans',
        'Enterococcus faecalis',
        'Escherichia coli',
        'Helicobacter pylori',
        'Lactobacillus gasseri',
        'Listeria monocytogenes',
```

```
        'Neisseria meningitidis',
        'Porphyromonas gingivalis',
        'Propionibacterium acnes',
        'Pseudomonas aeruginosa',
        'Rhodobacter sphaeroides',
        'Staphylococcus aureus',
        'Staphylococcus epidermidis',
        'Streptococcus agalactiae',
        'Streptococcus mutans',
        'Streptococcus pneumoniae']

genus = [x.split()[0] for x in mock]

family = ['Moraxellaceae',
          'Actinomycetaceae',
          'Bacillaceae',
          'Bacteroidaceae',
          'Clostridiaceae',
          'Deinococcaceae',
          'Enterococcaceae',
          'Enterobacteriaceae',
          'Helicobacteraceae',
          'Lactobacillaceae',
          'Listeriaceae',
          'Neisseriaceae',
          'Porphyromonadaceae',
          'Propionibacteriaceae',
          'Pseudomonadaceae',
          'Rhodobacteraceae',
          'Staphylococcaceae',
          'Staphylococcaceae',
          'Streptococcaceae',
          'Streptococcaceae',
          'Streptococcaceae']

```

```
rank = ['Kingdom', 'Phylum', 'Class', 'Order', 'Family', 'Genus', 'Species', 'Strain']

if __name__ == "__main__":
	fList = ['silva_spingo', 'silva_gast', 'rdp_spingo', 'rdp_gast']
	#f1_name = fList[1]
	#f2_name = fList[3]
	#f1 = file_parse(f1_name)
	#f2 = file_parse(f2_name)

	for i in fList:
		f1 = file_parse(i)
	#out = file('silva_rdp_gast.txt', 'w')
	#for i in sorted(f1.keys()):
	#	x = get_output(f1[i].tax, f1_name, f2[i].tax, f2_name)
	#	if not x == None:
	#		out.write("\t".join([i,x])+"\n")
	#out.close()
```

APPENDIX G
COLD LAND PROCESS FIELD EXPERIMENT CHANGE LOG GENERATOR

```python
from glob import glob
from datetime import datetime
import csv
import dbfread
import sys

class Version:
        unique_id = None

        def __eq__(self, other):
                if isinstance(other, self.__class__):
                        return self.unique_id == other.unique_id
                else:
                        return self.unique_id == other

class StratVer(Version):

        def __init__(self, headers, values):
                v = [i.strip() for i in csv.reader([values]).next()]
                self.data = {}
                self.load_strat(headers, v)
                self.unique_id = (self.data['PIT_NAME'], self.data['IOP'], int(self.data['TOP']), int(self.data['BOT']))

        def load_strat(self, headers, v):
                for i in range(len(headers)):
                        if i == 2:
                                self.data[headers[i]] = datetime.strptime(v[i], "%Y-%m-%d").date()
                        elif i > 2 and i < 14:
```

```
                if '.' in v[i]:
                    self.data[headers[i]] = float(v[i])
                else:
                    self.data[headers[i]] = int(v[i])
            else:
                self.data[headers[i]] = v[i]

class SummVer(Version):

    def __init__(self, headers, values):
        v = [i.strip() for i in csv.reader([values]).next()]
        self.data = {}
        self.load_summ(headers, v)
        self.unique_id = (self.data['PIT'], self.data['IOP'])

    def load_summ(self, headers, v):
        for i in range(len(headers)):
            if i == 3:
                try:
                    self.data[headers[i]] = datetime.strptime(v[i], "%Y-%m-%d").date()
                except ValueError:
                    self.data[headers[i]] = None
            elif (i > 3 and i < 7) or (i > 8 and i < 11) or (i > 11 and i < 34):
                try:
                    self.data[headers[i]] = int(v[i])
                except ValueError:
                    pass
                try:
                    self.data[headers[i]] = float(v[i])
                except ValueError:
```

```
                                        self.data[headers[i]] = v[i]
                        else:
                                self.data[headers[i]] = v[i]

def helper_factory( section, mode, arg1=None):
        if mode == "strat":
                if section == "names":
                        return ("/data/ice/shape_files/pit_iop_v2_strat.dbf", "strat")
                elif section == "key":
                        return "Key = (PIT, IOP, TOP, BOT)\n\n"
                elif section == "mapper":
                        return strat_col_map
                elif section == "table_ids":
                        return [(r['PIT'], r['IOP'], r['TOP'], r['BOT']) for r in arg1]
        elif mode == "summary":
                if section == "names":
                        return ("/data/ice/shape_files/pit_iop_v2_summary.dbf", "summary")
                elif section == "key":
                        return "Key = (PIT, IOP)\n\n"
                elif section == "mapper":
                        return summ_col_map
                elif section == "table_ids":
                        return [(r['PIT'], r['IOP']) for r in arg1]

def strat_col_map(index, mode):
        if mode == "ascii":
                if index == "DATE_":
                        return "DATE"
                elif index == "PIT":
                        return "PIT_NAME"
        elif mode == "shape":
                if index == "DATE":
                        return "DATE_"
```

```
                elif index == "PIT_NAME":
                        return "PIT"
        return index

def summ_col_map(index, mode):
        if mode == "ascii" and index == "DATE_":
                        return "DATE"
        elif mode == "shape" and index == "DATE":
                        return "DATE_"
        return index

def ice_import(category):
        if category == "strat":
                fn = glob("/data/ice/ascii/*strat.csv")
        elif category == "summary":
                fn = glob("/data/ice/ascii/*summary.csv")

        entries = {}
        for i in fn:
                f = open(i)
                ll = f.readlines()
                headers = [c.strip() for c in
    ↪ csv.reader([ll[0]]).next()]
                for j in ll[2:]:
                        if category == "strat":
                                sv = StratVer(headers, j)
                        elif category == "summary":
                                sv = SummVer(headers, j)
                        entries[sv.unique_id] = sv
                f.close()
        return headers, entries

def generate_logs(table_name, text_name, f):
        table = dbfread.DBF(table_name)
        table_ids = helper_factory("table_ids", text_name, table)

```

```
    headers, text_entries = ice_import(text_name)

    f.write("Shape Entries not in Ascii Files\n")
    for i in table_ids:
        if i not in text_entries.keys():
            f.write(str(i)+"\n")
    f.write("\n")

    f.write("Shape Columns not in Ascii Files\n")
    for i in table.field_names:
        if i not in [column_map(j, "shape") for j in headers]:
            f.write(i+"\n")
    f.write("\n")

    f.write("Ascii Entries not in Shape Files\n")
    for i in sorted(text_entries.keys()):
        if i not in table_ids:
            f.write(str(i)+"\n")
    f.write("\n")

    f.write("Ascii Columns not in Shape Files\n")
    for i in headers:
        if i not in [column_map(j, "ascii") for j in table.field_names]:
            f.write(i+"\n")
    f.write("\n")

    c = 0
    f.write("Modified Entries\n")
    for r in table:
        if text_name == "strat":
            i = (r['PIT'], r['IOP'], r['TOP'], r['BOT'])
        elif text_name == "summary":
            i = (r['PIT'], r['IOP'])

```

```
        if i in text_entries.keys():
            v = text_entries[i]
            start = True
            for j in table.field_names:
                if column_map(j, "ascii") not in v.data.keys():
                    pass
                elif r[j] == 'NoData' and v.data[column_map(j, "ascii")] == '-999':
                    pass
                elif r[j] != v.data[column_map(j, "ascii")]:
                    if start:
                        f.write("%s\n"%(str(v.unique_id)))
                        f.write("Column|\tShape|\tAscii\n")
                        start = False
                    f.write("%s|\t%s|\t%s\n"%(j, r[j], v.data[column_map(j, "ascii")]))
                    #print v.unique_id, j, r[j], v.data[attribute_map(j)]
                    c += 1
            if not start:
                f.write("\n")

if __name__ == "__main__":
    if "-strat" in sys.argv:
        table_name, text_name = helper_factory("names", "strat")
    elif "-summ" in sys.argv:
        table_name, text_name = helper_factory("names", "summary")

    change_log = "changelog_"+text_name+".txt"
    f = open(change_log, 'w')
```

```
        f.write("Change Log %s to ascii files\n"%(table_name.split('/')[-1]))
        f.write(helper_factory("key", text_name))

        column_map = helper_factory("mapper", text_name)
        generate_logs(table_name, text_name, f)
        f.close()
```

G.1 Versions as a Class

In the CLPX data, versions were encapsulated into individual classes so that computations between versions could be standardized and automated. The code could handle ingesting a variety of different formats of data while keeping the core comparison functions independent of individualized fields.

APPENDIX H
EARTH OBSERVING LABORATORY ANALYSER

```python
import csv
from datetime import datetime

class dataset:
        def __init__(self, d_id, d_title):
                self.versions = {}
                self.num = int(d_id)
                self.title = d_title

        def add_file(self, v_num, v_pub, v_crt, v_mod, f_name, f_crt, f_rev, f_notes):
                if v_num not in self.versions:
                        self.versions[v_num] = eol_ver(v_num, v_pub, v_crt, v_mod)
                self.versions[v_num].add_file(f_name, f_crt, f_rev, f_notes)

        def __repr__(self):
                out = ["%i: %s"%(self.num, self.title)]
                for i in sorted(self.versions.keys()):
                        out.append(self.versions[i].string(4))
                return "\n".join(out)

class eol_ver:
        def __init__(self, v_num, t1, t2, t3):
                self.num = v_num
                self.files = []
                self.v_pub = datetime.strptime(t1, "%Y-%m-%d %H:%M:%S")
                self.v_crt = datetime.strptime(t2, "%Y-%m-%d %H:%M:%S")
```

```
        self.v_mod = datetime.strptime(t3, "%Y-%m-%d %H:%M:%S")

    def add_file(self, f_name, t1, t2, notes):
        new_f = eol_file(f_name, t1, t2, notes)
        self.files.append(new_f)

    def string(self, indent):
        ind = ' '*indent
        out = [ind+"%s: %s    %s    %s"%(self.num, str(self.v_pub), str(self.v_crt), str(self.v_mod))]
        for i in self.files:
            out.append(i.string(indent+4))
        return "\n".join(out)

    def __repr__(self):
        out = ["%s: %s    %s    %s"%(self.num, str(self.v_pub), str(self.v_crt), str(self.v_mod))]
        for i in self.files:
            out.append("    "+i.__repr__())
        return "\n".join(out)

class eol_file:
    def __init__(self, f_name, t1, t2, notes):
        self.name = f_name
        self.f_create = datetime.strptime(t1, "%Y-%m-%d %H:%M:%S")
        self.f_revise = datetime.strptime(t2, "%Y-%m-%d %H:%M:%S")
        self.f_notes  = notes

    def string(self, indent):
        out = ' '*indent
        return out+self.name

    def __repr__(self):
```

```
        return self.name

def import_eol(fname):
    f = open(fname)
    f_reader = csv.reader(f, delimiter='|')
    f_reader.next()
    header = [i.strip() for i in f_reader.next()[1:-1]]
    f_reader.next()

    datasets = {}
    for i in f_reader:
        row = [j.strip() for j in i[1:-1]]
        if len(row) < 10:
            continue
        if int(row[0]) not in datasets:
            datasets[int(row[0])] = dataset(int(row[0]), row[1])

        datasets[int(row[0])].add_file(row[2], row[3], row[4], row[5], row[6], row[7], row[8], row[9])
    return datasets

if __name__ == "__main__":
    filedir  = "/data/EOL/"
    #filename = "dataset_files_version_metadata.txt"
    filename = "EOL_dataset_version_metadata.txt"

    datasets = import_eol(filedir+filename)
    print len(datasets.keys())

    num_ver = {}
    for i in datasets.keys():
        how_many_versions = len(datasets[i].versions.keys())
        if how_many_versions not in num_ver:
            num_ver[how_many_versions] = 1
        else:
```

```
                    num_ver[how_many_versions] += 1
        for i in sorted(num_ver.keys()):
                print i, num_ver[i]
```