LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-729885

Scalable Metadata Management for a Large Multi-Source Seismic Data Repository

J. M. Gaylord, D. A. Dodge, S. A. Magana-Zook,
J. G. Barno, D. R. Knapp

April 24, 2017

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Lawrence Livermore National Laboratory

Scalable Metadata Management for a Large Multi-Source Seismic Data Repository

J. M. Gaylord, D. A. Dodge, S. Magana-Zook, J. Barno, and D. R. Knapp

April 28, 2017

1

Disclaimer

Abstract

In this work, we implemented the key metadata management components of a scalable seismic data ingestion framework to address limitations in our existing system, and to position it for anticipated growth in volume and complexity. We began the effort with an assessment of open source data flow tools from the Hadoop ecosystem. We then began the construction of a layered architecture that is specifically designed to address many of the scalability and data quality issues we experience with our current pipeline. This included implementing basic functionality in each of the layers, such as establishing a data lake, designing a unified metadata schema, tracking provenance, and calculating data quality metrics.

Our original intent was to test and validate the new ingestion framework with data from a large-scale field deployment in a temporary network. This delivered somewhat unsatisfying results, since the new system immediately identified fatal flaws in the data relatively early in the pipeline. Although this is a correct result it did not allow us to sufficiently exercise the whole framework. We then widened our scope to process all available metadata from over a dozen online seismic data sources to further test the implementation and validate the design. This experiment also uncovered a higher than expected frequency of certain types of metadata issues that challenged us to further tune our data management strategy to handle them.

Our result from this project is a greatly improved understanding of real world data issues, a validated design, and prototype implementations of major components of an eventual production framework. This successfully forms the basis of future development for the Geophysical Monitoring Program data pipeline, which is a critical asset supporting multiple programs. It also positions us very well to deliver valuable metadata management expertise to our sponsors, and has already resulted in an NNSA Office of Defense Nuclear Nonproliferation commitment to a multi-year project for follow-on work.

Introduction

The Lawrence Livermore National Laboratory (LLNL) Geophysical Monitoring Program (GMP) has long maintained a database of seismic waveforms and supporting metadata. In the early 1990's we had only hundreds of stations in our waveform database and fewer than a million segments. But, as of late 2016 we have more than 9600 stations with waveforms and hundreds of millions of segments in our archive (Figure 1).

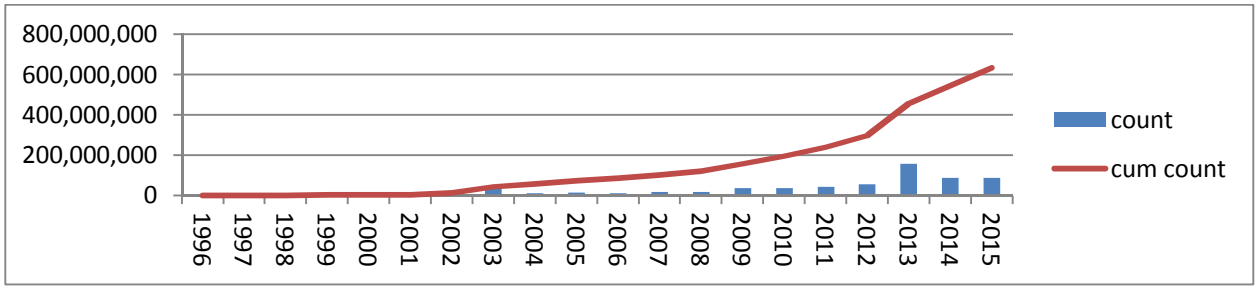

Figure 1 Incremental and cumulative waveform segments ingested into the GMP archive by year

Having large amounts of seismic data is both an opportunity and a challenge. It allows analyses that were impossible a few years ago, but at the cost of introducing significant data management complexities. In current large-scale data analysis efforts, scientists must spend considerable time resolving metadata issues before starting scientific analysis. The GMP data repository and associated suite of customized Java data ingestion tools helps insulate this responsibility from the scientists, but it is becoming increasingly difficult to manage as the number of included stations and sources grows.

As with other sensor data, seismic data is only useful when there is an accurate understanding of the physical channel that produced it, including information about the channel's location, hardware, operation, and epoch of operation. This critical information can be missing, in unexpected physical units (e.g. m/s vs nanometers/sec), recorded with too little precision, inconsistent with other data from the same source, indeterminate for a specific point in time, wholly in error, or subject to any number of other problems.

Integrating data from multiple sources poses additional challenges. Naming conventions are not standardized and there are no universal identifiers for seismic stations. Null and infinite values are often ad hoc. Formats and units of measure are often inconsistent between data providers. Time can be recorded in different units (e.g. days, seconds, or fractional seconds), formats, and time zones. Other unit and precision differences between sources introduce further uncertainty.

We have developed a suite of Java data ingestion tools to mitigate these issues through complex transformations as data are ingested. Over time, however, the tools have become unwieldy and brittle after decades of enhancements to accommodate increasing data drift and

variation. We are also realizing that our current metadata schema is insufficient for provenance tracking and inflexible to new data sources and formats. We too often resort to handling metadata issues with manual intervention, which limits scalability, reduces repeatability, and introduces additional data uncertainty. In short, our data pipeline is becoming significantly less effective as our data reach increases.

Commercial industries now routinely depend on customer analytics derived from massive quantities of diverse and unstructured data. Over the last decade technologies like Hadoop and sophisticated analytics engines have been developed to help process data, but there is still the complexity of moving data into the data warehouse and organizing it for analysis. This has created a new market, for which dozens of proprietary and open source data flow tools have been developed. However, adapting them for processing seismic sensor data is complex since they were developed around commercial domains and mostly textual data. Also, the technologies are rapidly emerging and evolving with little coordination, proprietary tools are expensive, and there is very little documentation or support for open source tools.

Our goal in this project was to overcome these complexities and leverage available technologies to design and prototype a more flexible, scalable, and maintainable seismic data ingestion system for GMP. We started by surveying leading open source data flow tools and assessing them for the seismic domain and our workloads. Next, we developed a unified schema for multi-source seismic metadata to overcome limitations of existing schemas. We then produced a conceptual four layered design for the overall system, defined interfaces between the layers, and developed initial implementations of key components in each layer. A variety of seismic source data ranging from large temporary field networks to openly available web services was used to test functionality. Finally, quality metrics were developed to evaluate incoming metadata, advise metadata transformations, and describe the final metadata results for use.

For this report, we will often refer to seismic parametric metadata as simply “data”.

Tool Survey

Our focus for this project was metadata ingestion, but we wanted to ensure the resulting framework would also eventually accommodate waveform ingestion and data management. For metadata, the challenge is variety and veracity. For waveform data, the challenge is primarily volume.

Ideally, we would replace all our existing ingestion codes with one tool that would handle our metadata ingestion pipeline from the data source to both a Hadoop (Vance, 2009) data store for analytics and the Oracle database used by our users and applications. This ideal tool would ease maintenance overhead and limit the need for custom code by abstracting away as much of

the nuts and bolts of the infrastructure as possible, and provide configurable interfaces for all our intended data sources and sinks out of the box. A full-featured and intuitive Application Programming Interface (API) for easy set-up and operation with our existing technology stack would also facilitate a rapid start.

A survey of Big Data conference presentations showed an emerging standard set of utilities from the open source Hadoop ecosystem for ingesting Big Data into a data warehouse. Apache Kafka (Primack, 2015) is the ubiquitous choice for getting data into the cluster, Apachie Oozie (Islam and Srinivasan, 2015) is a common data workflow manager, Apache Storm (Anderson, 2013) and Apache Spark (Zaharia, 2016) are used for onboarding streaming data, and Apache Sqoop (Jain, 2013) is the primary interface to external databases. We have experience with Spark and Sqoop, and enlisted a summer student to research Kafka and Oozie. Although each of these tools provides necessary functionality for data ingestion, none of them appeared to be the holistic solution we need.

Many companies with mature Big Data warehouses operate data ingestion pipelines built from these low-level utilities, but we (and many others) have found this to be a non-trivial approach with substantial complexity and maintenance overhead. This reality has led to a growing market for end-to-end data management products that abstract away as much of the lower level complexity of constructing a data pipeline as possible. We started this project by surveying this space to borrow proven code and simplify our overall process as much as possible.

We eliminated proprietary tools to avoid prohibitive license fees and their more prescribed designs, which are too often specifically aimed at commercial applications. In the open source space, we sought out stable, proven, enterprise grade technologies with stable releases, and ignored emerging options or those with smaller or less active user or developer communities. This logic quickly eliminated all candidates except Apache NiFi (Bridgwater,2015), a data ingestion management tool.

Apache NiFi

NiFi is a Java framework and web interface developed by the NSA for managing data movement between systems (NSA, 2014). NiFi was open-sourced under the Apache license in 2014 and quickly became a top-level project. It provides end-to-end data pipeline automation and a flow-based programming model. It was designed to fill the gaps between lower level data movement utilities, and more specifically to improve security, interactivity, scalability, and traceability. It includes a library of over ninety pre-built modules for specific data movements and transformations, and is easily extended and customized with Java code. Data ingestion paths are built as directed graphs using drag-and-drop components in a web interface, or with an

XML configuration file. Once a pipeline has been set up and started, the graphical interface can also be used to display and manage the pipeline during operation. The GUI display includes data flow rates, the state of data buffers between processing points, and the status of error streams. A particularly valuable feature of NiFi is its built-in provenance tracking system, which generates and stores metadata about each processing step for each data file or record that goes through the system. NiFi is deployed on one or more edge nodes of a Hadoop or other storage cluster.

NiFi Experiment with Seg-Y Data

Initial tests of NiFi functionality for our most straight forward data ingestion cases were promising, so we decided to test it further by developing a flow for a more problematic dataset. We had recently received data files from a large temporary network deployed for a Source Physics Experiment shot. The files were in Seg-Y format (Norris, and Faichney, 2002), a commonly used but loosely specified binary format for exchanging seismic trace data. The format and size of the files were challenging, and gave us a practical use case for defining a more complex workflow in NiFi. It also gave us an opportunity to establish a team development process for NiFi using containers (discussed in the Appendix), and better understand the level of effort required to create an operational data processing system within the NiFi framework. We used the Seg-Y exercise to determine the flexibility of NiFi for handling problem data, assess its usefulness for processing binary formats, and study the practicality of integrating existing data ingestion tools into a NiFi workflow.

Initially we attempted to use as many of the built-in processing tools provided by NiFi as possible. This worked well for some of our data flows but quickly proved problematic for others. Like many Big Data frameworks, NiFi was originally targeted for processing textual data, and thus has very few built-in components for interacting with binary data. Standard Seg-Y files have many waveform and survey related metadata fields but the payload is entirely binary, so we were forced to extend the pre-built library with custom components for this application.

Fortunately, NiFi can treat communicating with and running existing tools as just another "processor" directive in a flow. Additionally, NiFi has a relatively robust framework for writing plugins that can be installed directly into the NiFi server natively to operate with the DataFlow graph interface. By using these two capabilities in concert with the existing NiFi processors we developed some proof of concept data flows for handling Seg-Y files under a variety of circumstances.

Custom NiFi processors can be written for specialized processing while retaining the benefits of the NiFi ecosystem like complex flow routing, automatic backpressure, etc. We took advantage

of this extensibility to extract metadata contained in our binary files and record structured JSON representations automatically with the custom processor flow shown in Figure 2.

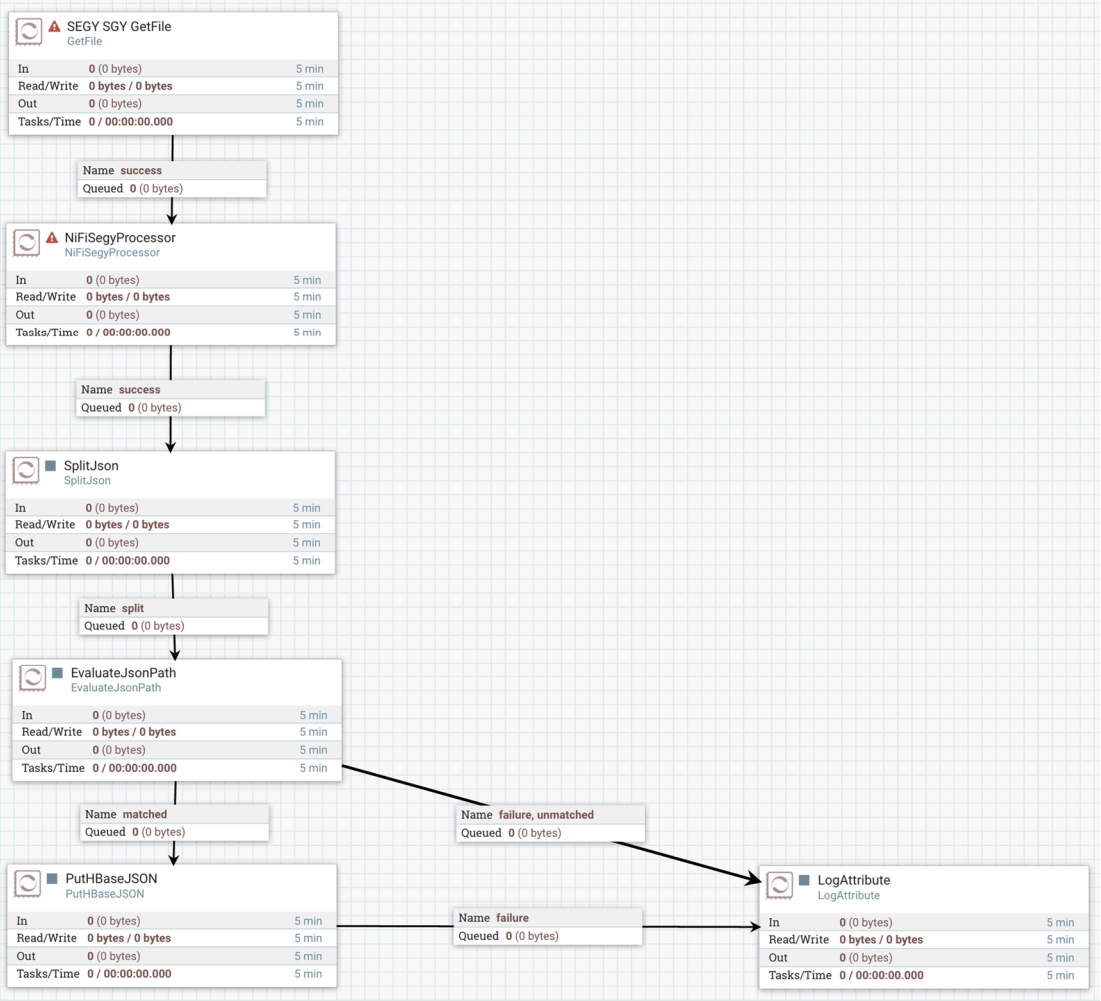

Figure 2 A simple data flow using built-in NiFi processors and a custom NiFiSegyProcessor plugin

This exercise confirmed that NiFi offers many valuable advantages for complex data flows, but it eventually exposed some drawbacks. Computations with exceptionally complex, long running, or blocking operations can introduce bottlenecks in the ingestion pipeline. In addition, NiFi is not designed to handle cases where additional information is needed to process the incoming data, or where something about a previous data object must be known to process the current one.

In general, NiFi follows a highly scalable “stateless” paradigm where ingestion flows are processed in isolation from other data sources. It is geared towards extracting information from objects rather than processing that requires operations such as Structured Query Language (SQL) “join” statements that merge two datasets together. The NiFi community has

started to discuss including functionality for more stateful workloads in future development to allow these data flows to integrate more tightly into the NiFi ecosystem.

Integrating External Tools

A current solution to the isolated flow problem is to use NiFi as a flow controller to and from external processing pipelines and tools (Figure 3). In this case, the stateful processing code is handled in external applications which are called and tracked by an end-to-end NiFi flow. Using an external tool as a NiFi processor is also a useful mechanism for integrating existing data processing tools into a NiFi flow. However, when designing such a flow extra attention must be given to scaling, and how NiFi communicates with and manages processes.

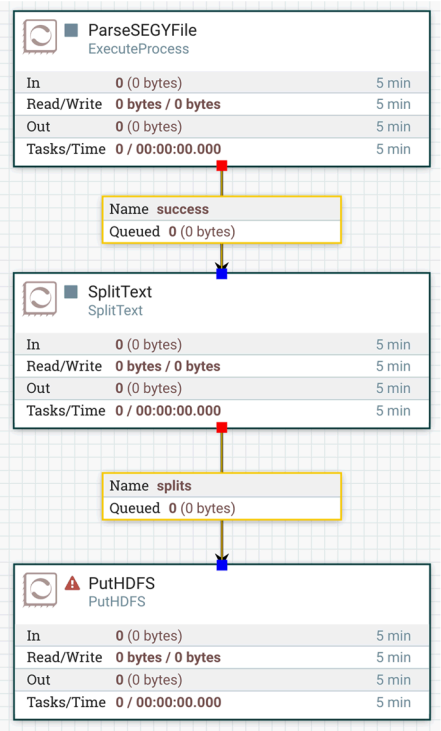

Figure 3 Calling an external program (ParseSEGYFile) and accepting its output as part of a flow

The ability to manage data flow to and from external processes also allows for long running or asynchronous operations to exist as terminal input and output sub-flows in the overall NiFi flow. This can resolve many more complicated data processing problems but is architecturally more complex, and can result in sacrificing some of NiFi's built in flow control and data provenance capabilities.

Our conclusion from this exercise was that even though the built-in NiFi processors fell short for parsing binary data or handling stateful tasks, NiFi was flexible and extensible enough for us to create a viable solution despite this. Our data rates are not excessive, our ingestion is not

expected to be real-time, and the added overhead of the work-arounds described previously is acceptable. For our use cases NiFi's efficiencies easily made up for the added complexities of our stateful and binary ingestion workloads with useful data management and monitoring features such as a centralized interface for flow control and automated data provenance. Overall, we found NiFi to be a very capable platform for automating and managing our data ingestion pipeline.

Unified Schema

With the flow framework selected and validated, we turned our attention to specifying the minimum required set of data elements needed to represent all the metadata types we process. The schema for our existing seismic data archive is based on the Center for Seismic Studies version 3 (CSS3.0) database schema (Anderson et al. 1990). This format has been a standard in the monitoring community for years, and it has served us well. Many other seismic data formats exist, however, and we must integrate such data into our schema to support our research objectives. Often the results are less than one would hope for.

Many of our data sources adhere to the newer Standard for the Exchange of Earthquake Data (SEED) data standard (IRIS, 2012). This schema does not reconcile well with our CSS schema, and critical pieces of information may lose fidelity or be lost completely when we force SEED source data into our database. Addressing this problem is a major goal of this project, and it requires that we adopt a new data schema that preserves and reconciles the necessary data elements from the CSS and SEED standards, as well as any others that we have processed in the past or may need to process in the future.

Our approach in designing a new schema was to start by preserving the necessary elements in both CSS and SEED at the highest fidelity available. We then augmented this super set with additional provenance tracking fields and other data elements we found to be missing in both standards. We refer to the result as the "unified schema".

Schema Description

The unified schema is a hierarchical data structure that gives primary importance to the source of the data. Within each source the network, station, and channel names are assumed to be unique and consistent with physical locations. No attempt is made to reconcile naming conventions between sources. This allows a major data processing simplification compared to CSS, which falsely assumes consistent naming across all data sources, and SEED, which assumes consistent naming within networks. The primary keys for our tables include all the parent table codes starting with the source code and both the begin-time and end-time of the epoch. A

surrogate key identifier is added to each row after ingestion is complete to support efficient joins in downstream applications. Joins on these integers are much more efficient than the joins on composite text keys that would otherwise be necessary. The basic schema is depicted in Figure 4 below.

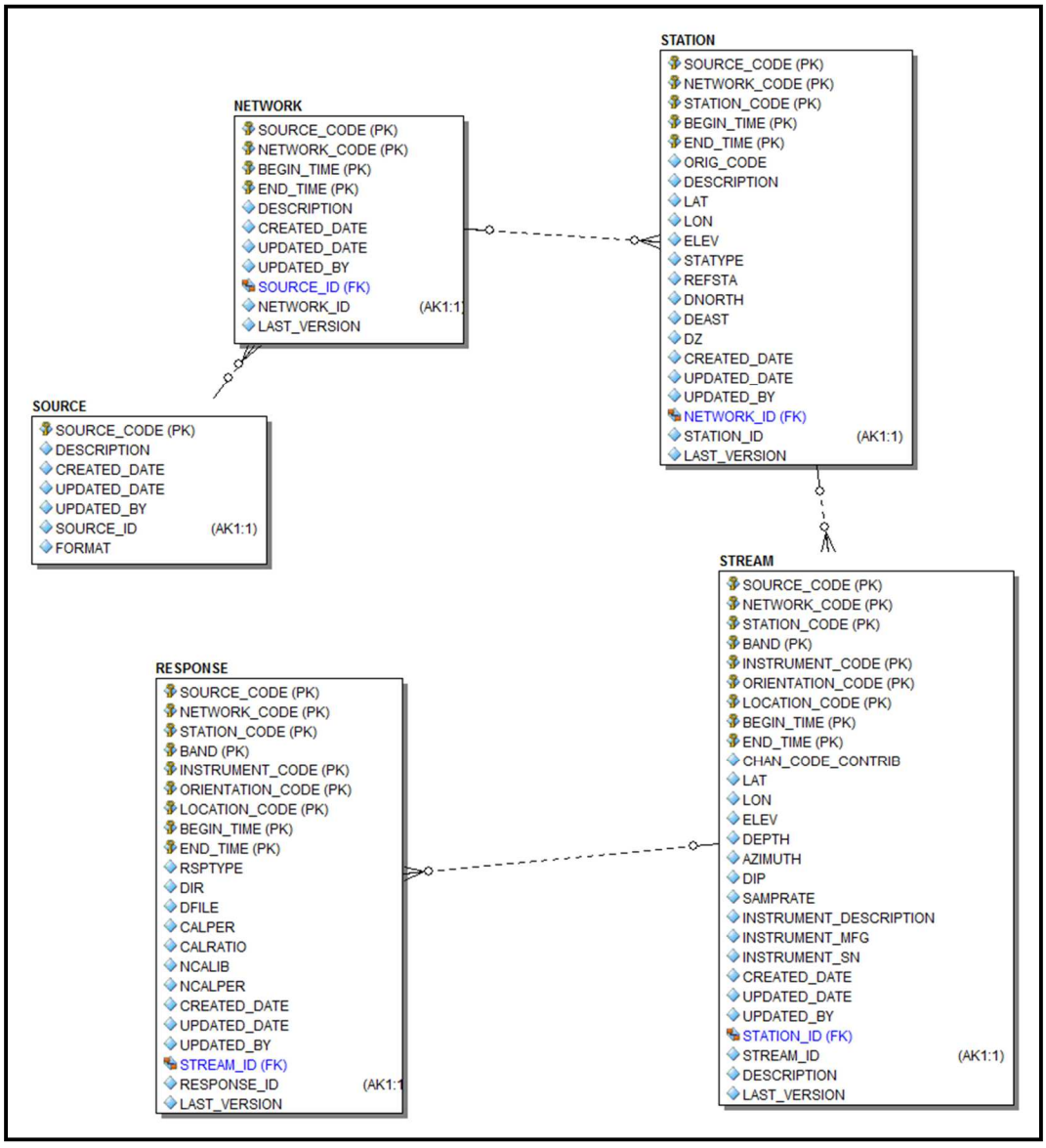

Figure 4 The Unified Schema

In this schema, a stream is a new construct intended to represent the lowest level of metadata available for a specific data channel. A stream record represents a sensor epoch whenever sensor level data is available, otherwise it represents the channel epoch. We did this because the details of the channel epoch are frequently a source of date errors, and are not as important as sensor epochs for determining the correct response and interpreting waveform data. Channel codes are broken into their individual band, instrument, orientation, and location

components. Streams may have zero, one, or more associated response files depending on what is supplied by the data source.

The columns of the unified schema are drawn from The International Federation of Digital Seismograph Networks (FDSN) (Romanowicz, 1990) SEED version 2.4 reference manual and from the CSS version 3.0 database schema specification. In SEED, stations are identified by network code, station code, and time of operation. Those keys are retained here and augmented by a source code that identifies the organization from which the data were obtained. In CSS, although stations may be associated with a network, they are not required to be.

We also adopted the SEED convention of representing all date-times as precisely as the data allow. In CSS, date-time data is sometimes represented as epoch times and sometimes as an ordinal date (YYYYDDD). Although the ordinal date representation is often convenient, mixing the two representations can result in temporal database inconsistencies, e.g. sensor epochs that straddle channel epoch boundaries in CSS.

We included the CSS columns (STATYPE, REFSTA, DEAST, and DNORTH) in the station table to retain the array information present in data from the US NDC SITE files. Even though SEED blockette 35 provides information about beams, station XML apparently does not. And, while the CSS representation of arrays is problematic, we needed a place for this information in our input tables. Our unified schema includes a new representation for array data that removes the limitations in the CSS representation.

The STREAM table in the unified schema is based on SEED blockette 52. As with the STATION table, we have added a source code as part of the key. Another important deviation from both SEED and CSS practice is that in addition to the provided channel code, STREAM has columns for BAND, INSTRUMENT, and ORIENTATION. We introduced these columns to allow the final (integrated) STREAM table to be in first normal form (1NF). Channel code is often treated as atomic, and therefore suitable as a database column. But in SEED usage it is the concatenation (BAND-INSTRUMENT-ORIENTATION) and is thus expressing 3 facts. Some programming logic is complicated if the only access to those facts is through channel code.

Unfortunately, some legacy data does not follow the SEED channel naming convention and cannot be easily decomposed. For example, data with the channel code “sz” can be used to set BAND and ORIENTATION, but what about channel “uu23”? To accommodate these, we rely on the provided channel code (CHAN_CODE_CONTRIB) with the expectation that it will be used to support queries where only the name matters.

Responses are, in a sense, additional information about a STREAM. However, Streams can have 0 to N responses. Therefore, we broke RESPONSE out as a separate table. It has the same keys

as STREAM (although the times are specific to the response epochs). The fact columns (RSPTYPE, DIR, DFILE, CALPER, CALRATIO, NCALIB, and NCALPER) are from CSS.

Inserting New Data

In the unified schema, every new epoch specification received from each source is kept indefinitely, along with additional metadata about the version, create date, and date last updated. New data updates can only overwrite identically keyed epochs. This updates the non-key attributes, the version number, and the last updated date, but does not affect any other rows. This is to ensure that newer data persists, but without eliminating previous versions of epochs which may still be valid. If exact key matches are not found, a new row is inserted with the current timestamp and version number one. The result is a cumulative set of data by source where newer data is merged into existing data based on matching keys.

Layered Architecture

Our current database could conceivably be modified to accommodate this new unified schema design except that our current architecture is fundamentally inflexible to significant change. This is primarily because we ingest data directly into the same schema that is used by our analysis code and users, so a single field change means every access path that uses the field must be upgraded concurrently. In addition, critical key values such as source codes and network codes would need to be inferred from the other data or filled with a default value since no provenance records from ingestion exist.

Another primary goal of this project was to remove this inflexibility to change. All indications point to continued volatility in incoming seismic data formats as new technology makes exponentially more deployments a reality and as we increase our data appetite accordingly. The layered architecture depicted in Figure 5 is specifically designed to support data drift by decoupling the environments used for raw storage, ingestion, transformations, and presentation from end users and applications. This allows each major pipeline function to evolve independently as necessary. Changes in any layer will not impact any other or downstream data consumers, if the data interfaces between layers are maintained.

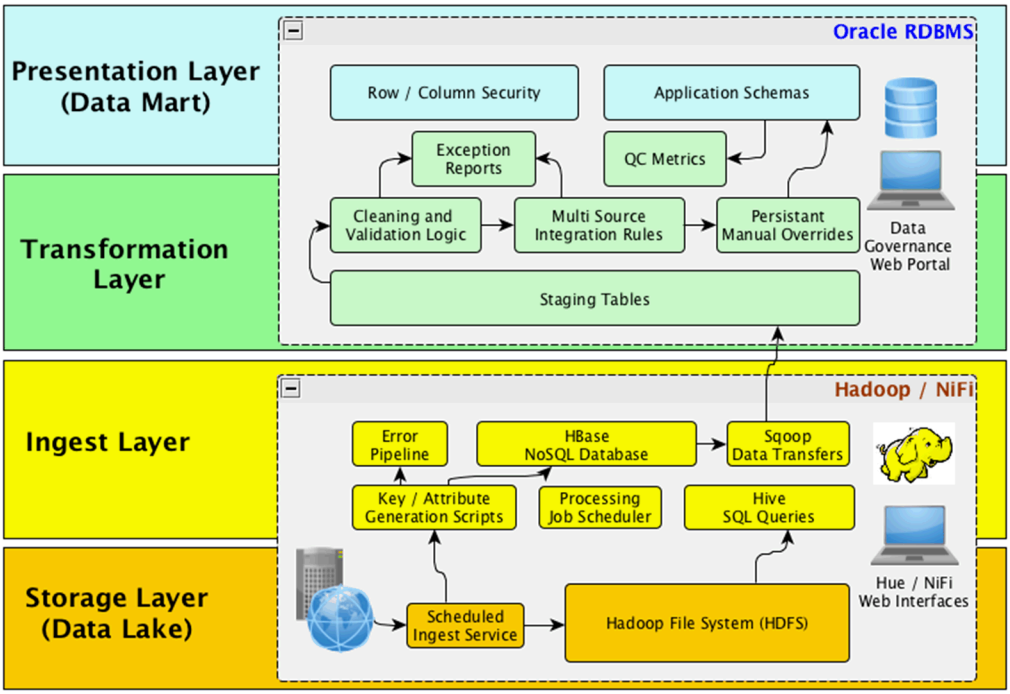

Figure 5 An early conceptual diagram of the layered architecture for the data pipeline

Data enters the pipeline through one or more fetching tools in the storage layer. Next, the ingestion layer normalizes the units for each field among the various data sources, performs minimal quality checks, and presents a unified schema to the transformation layer. In the transformation layer, more rigorous quality checks screen out problematic records resulting in a "Data Mart" ready for presentation. The presentation layer may include anything from a simple database schema or data service to complex analytic tools.

Layer 1: Storage

The storage layer is the first stop in the data pipeline and it provides the initial landing area for all new data coming into the system. The incoming data is stored in its original raw form without any transformations or other processing. This type of repository is referred to as a "data lake", and differs significantly from the "data mart" landing archive currently used by us and many others.

A data mart stores curated data which is assumed to be consistent. Unfortunately, the data we receive is not necessarily consistent internally or with respect to equivalent data from other sources, so the data stored in the mart may be a significantly modified or trimmed version of what was originally received. Our data needs change over time, and once the data has been processed into the mart we are often no longer able to infer the original data accurately enough to reprocess it correctly. This leads to inflexibility in the system and stale or unreliable data in our curated set.

Another source of inconsistency occurs when a data source changes, deletes, or updates values and relationships in their metadata after we have ingested it. Since we modified the data once already in a previous ingestion run based on the original version of the data, in the subsequent ingestion run we may not be able to accurately match up the new version with the rows already in our database. After the ingestion of the new data we quite possibly end up with multiple inconsistent versions of the same data in our data mart. A related type of inconsistency can occur if the processing logic in our ingestion code has changed between processing runs.

These problems are common and they should be expected. They are not due to a lack of discipline at the data source or unstable ingestion software at our end. A main cause of the problems is that as the number of data providers has grown, so has the number of different formats for providing data, as well as the duplication or overlap of data between sources. This leads to significant ambiguity in units of measure, points of reference, labels, naming conventions, time conventions, precision, and more. When we first built our models for ingestion, we had a limited number of data providers who in turn had a limited data set. Not only has the number of data providers grown, so have their holdings. We have tried to organically grow our monolithic data ingestion process to meet this and our expanding needs, but it is starting to show significant cracks and limitations and is becoming increasingly difficult to maintain.

Data Lake Configuration

For our initial prototype data lake we implemented a simple directory structure on a Network File System (NFS) mount. The template for the directory structure is:
…/dataLake/<data provider>/[metadata|waveforms]/<Data format_ batch timestamp>/…./<file>

Three data source providers were used in the sample set:

- US National Data Center (USNDC)
- IRIS
- UNR

As the examples show, data ingestion from each data source can be scheduled separately and with a different frequency.

Examples of the meta-data directories:

./USNDC/metadata/CSS3.0_17Mar2016T04.30.00-0700
./IRIS/metadata/StationXML_22Jun2016T15.29.13-0700
./IRIS/metadata/StationXML_22Jun2016T15.29.19-0700
./IRIS/metadata/StationXML_24May2016T11.00.00-0700
./UNR/metadata/SEED_01Apr2016T23.59.59-0700
./UNR/metadata/CSS3.0_06May2016T23.59.59-0700
./UNR/metadata/CSS3.0_18Apr2016T23.59.59-0700
./UNR/metadata/SEED_18May2016T23.59.59-0700
./UNR/metadata/SEED_02May2016T23.59.59-0700

A transaction log was kept for each data source. The purpose of the transaction log was to identify when and where files from data sources were introduced into the data lake.

An example of the UNR transaction.log:

==========

06/28/16

Copied the following from /SPE34/SPE/2016/. These were originally rsynced from UNR by Terri.

css3.0_31May2016:

nnss nnss.lastid nnss.sensor nnss.sitechan nom_response

nnss.calibration nnss.network nnss.sensormodel nnss.snetsta response

nnss.instrument nnss.schanloc nnss.site nnss.stage

seed_31May2016:

basement nnss_dataless_seed_fullres rdseed.stations resp

For our prototype implementation, we used two different methods for acquiring data. For USNDC and UNR, we utilized the existing tools such as ftp and robo-copy. For IRIS, we utilized a tool that could be used in the pipeline in an automated manner, Apache Nifi, to perform the data fetching and data routing tasks (Figure 6).

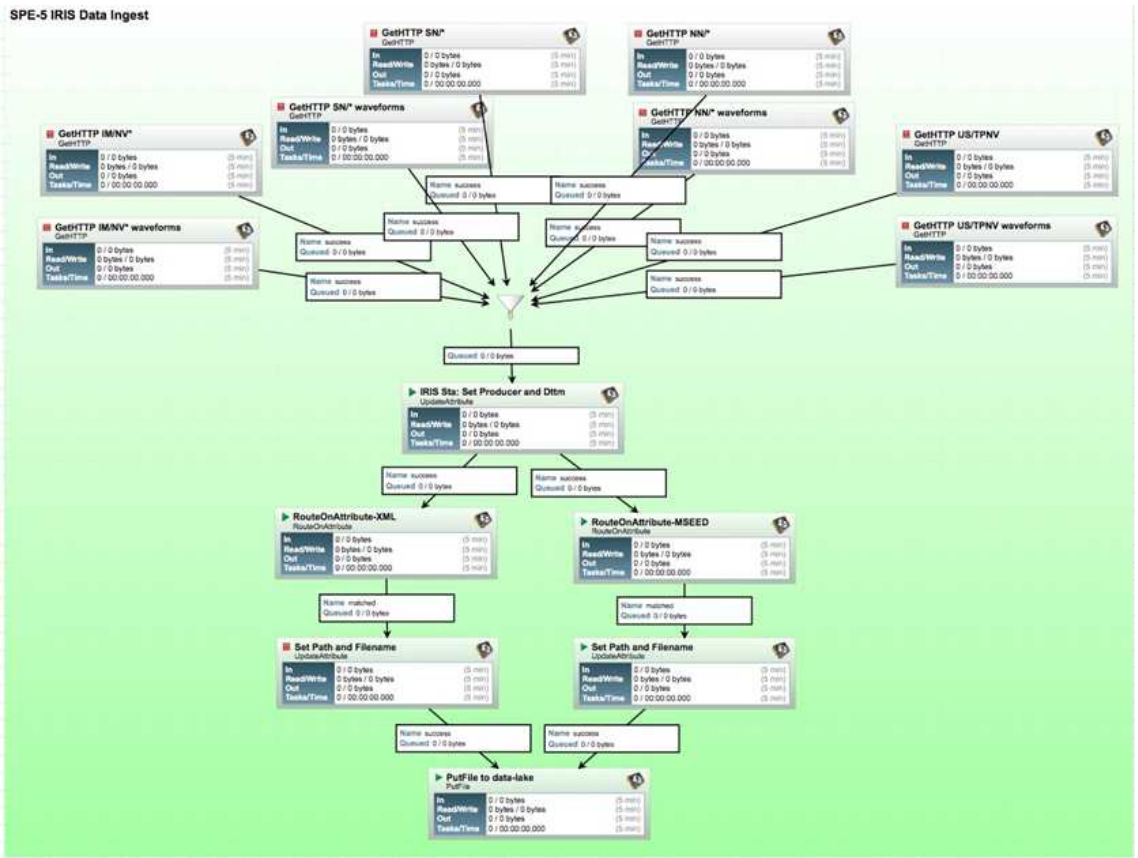

Figure 6 NiFi IRIS metadata workflow

Discussion

The automated NiFi solution has several advantages over the manual method. NiFi can be scheduled to run at times that take advantage of under-utilized computational and network resources without human intervention. Logging for the runs is also fully automated. By automating these tasks, the data curator is freed up from these mundane ingestion tasks and can focus on the more difficult challenges of data management.

The system just described is only a start, and there is more work to be done. We plan to reduce the rigidity of our data collection processes with a framework that allows the directory structure to be self-descriptive. For example, XML or JSON configuration files for each data source would allow a wider range of tools to interact with the source data, and provide flexibility in how the data is stored in our archive. Managing data from different source providers would be configurable and extensible instead of hard-wired and uniform.

The storage layer provides the directory structure to subsequent processing steps for locating data elements. Instead, we could use a "query service" style interface to provide this and other information. For example, the service could be asked for only the new elements added after a certain time, or only the elements that meet other provenance criteria not encoded in the directory name. As the data lake grows and becomes more complex these types of features will become necessary.

Until it becomes standard for data sources to provide a change log for their data, the storage layer will need to request all data. This will often result in data that is already in the archive being returned by the data source. The storage layer must have capabilities to de-duplicate identical data (especially waveforms) so only one copy of each version is stored in the data lake. At the same time, the de-duplication process must not alter the original form of the data.

We have not yet determined where to put the data lake to provide the most scalability, economy, and efficiency. The system is designed to keep all versions of data ever received from each source to support reproducibility for publications, data forensics, and data recovery. Currently, the incoming data are stored in both the Hadoop Distributed File System (HDFS) for scalability and our NFS for ease of access. This means four copies of each raw file are retained since HDFS keeps three by default for redundancy and performance. Before we add significant waveform data to this solution, we will need to develop a more efficient storage plan.

Our ingestion system issues requests to get new data from data providers. A future opportunity to consider is enabling the storage layer to support requests initiated by a data provider as well.

Future Plans

In the future, we will put access to the data lake behind an API to abstract out the concrete implementation underlying our storage. Notionally whenever a file is placed into the lake a unique identifier is generated that associates it with all data products created using that file in the processing pipeline. This allows for downstream processes to be traced back to the source material, and provides a means to re-process the raw data if necessary.

At least two additional components beyond the basic file storage system are necessary to serve this API: a metadata repository to hold the identifier for storage location mapping and some stable service to act as the communication mechanism.

The unique identifiers should be 'stable' in that they are re-creatable should the metadata storage fail or otherwise become corrupted. The metadata storage and communication mechanisms also need to be inherently scalable to the same order as the underlying storage mechanism used to hold the actual files. Both topics are subjects of ongoing research and will require experimentation but a simple prototype we are presently working on is diagramed in Figure 7.

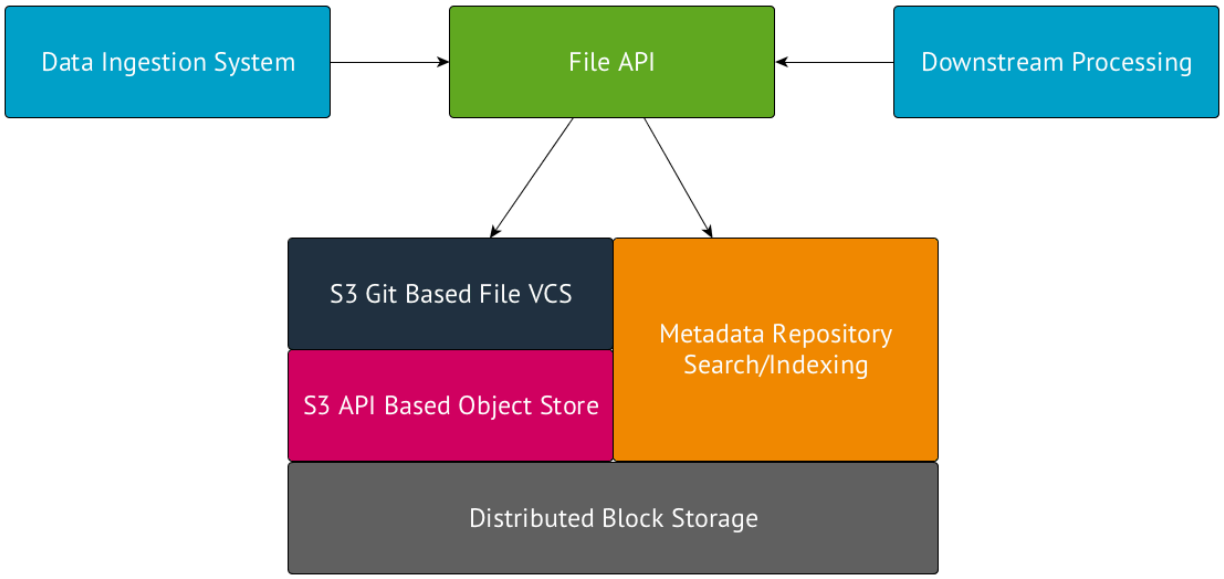

Figure 7 - Simple diagram of one possible implementation for the data lake

Layer 2: Ingestion

Design Goals

The ingestion layer is responsible for taking the items in the data lake, packaging them into a format that is easily consumable by Big Data tools, performing schema validations as a first pass

at quality checks, and outputting the results in a unified format. An important part of creating the unified schema is organizing data from multiple ingestion runs and sources into one set which can include multiple versions of data from each source. As different data providers use different units for their measurements, these must also be unified as part of the ingestion process. Data from this layer is exported to the transformation layer for further cleaning and analysis.

Work Done

Choice of Intermediary Formats

The choice of file format to hold intermediary data in the ingestion layer depends on many factors including technical limitations of processing software and intended use-cases for the data. It can have significant ramifications for storage hardware, the software stack, and application performance. We tested a variety of file formats to assess their efficiency with data processing tools implemented in Spark (Zaharia, 2016) for performance and scalability.

Raw/Native File Format

The simplest and most efficient choice from a storage perspective is to process the files from the data lake without any conversion in between. Although our cluster compute nodes can access raw files in the data lake via the network we find pre-packaging multiple raw files into an intermediary aggregate format provides many advantages.

Going from thousands of files whose size is on the order of kilobytes to a few files whose size is on the order of gigabytes or larger provides an immediate efficiency boost to input and output (I/O) rates. Additionally, by storing these large consolidated files in HDFS, we can leverage data locality for scalable subsequent processing instead of hitting a centralized file server on every request, such as with NFS.

Apache HBase

Apache HBase (George, 2011) is a Hadoop distributed NoSQL store. NoSQL is a class of relaxed or limited SQL data stores that promise scalability and extensibility. HBase is intended to host very large tables e.g. billions of rows by millions of columns) using Hadoop and HDFS. Unlike pure HDFS, which is append only, it supports record level inserts, updates, and deletes. Like many other technologies in the Hadoop ecosystem, HBase is optimized for certain types of use cases and workloads. We found it to be prohibitively burdensome and non-performant for this application but suitably flexible and extensible for others.

HBase abstracts records as key-value pairs, storing both key and value as an ordered list of byte arrays. We quickly learned that the key choice is critical to the functionality and performance of the store. One approach is to create a composite key that includes the fields and conditions most commonly used for querying. Querying on fields not in the key results in each record

being de-serialized to check the query condition, which can erode performance to the point of making nodes unresponsive. Care must also be taken to ensure composite keys are always unique. Another strategy is to use a hash or other numeric unique identifier as the key. This is easy to manage and is performant if the key is easily known in advance for data requests.

Our initial HBase implementation combined the waveform into the HBase record with a composite key. In this case, HBase was far slower in I/O throughput compared to reading directly from HDFS for our data sets. In addition, the keys were complex, difficult to choose effectively, and hard to manage. This approach for storing waveform segments was abandoned in favor of HDFS file formats like Avro.

In our current implementation, we use HBase to store metadata only records with unique numeric keys in a query-able data catalog. This is looking to be a very suitable application for HBase and illustrates some of its advantages. One is the column family format, which allows different fields to be defined for different records. This lets us store multiple data formats in one table and a highly flexible schema without trimming or transposing the original fields and values. Another useful feature of HBase is the automatic versioning of records. This preserves a configurable number of the most recent values for each cell over time for point in time queries. More work is needed to test the performance of this implementation at scale with complex data formats, multiple data versions, and our full range of query workloads. This approach only manages the metadata and a pointer to the waveform payload, which is stored separately.

Apache Avro

Avro (Russell and Cohn, 2012) is a serialization framework used in the Hadoop project. Part of the Avro specification is the Object Container File container file format in which you can bundle data from what would otherwise be too many small files for HDFS to handle. We use the term “Avro” in this report to refer to the container file format.

Avro is not bound to a key-value scheme, but Avro files can still be queried using SQL directly in Spark. This does not use Region Servers and some other daemons HBase requires, and therefore eliminates a category of problems that affected job success and performance for our waveform storage workloads. Avro also provides row level compression and has support for schema evolution.

Ultimately, however, Avro turned out to be much less efficient than other solutions for data exploration use-cases and other queries that care only about a subset of columns. Its row-based format means the entire row is always retrieved, even if only certain fields are needed. We switched to a columnar store to avoid this performance bottleneck and to maintain compatibility with current direction of Spark development.

Apache Parquet

Apache Parquet (White, 2015) is a relatively new format to the Hadoop ecosystem, but its adoption as a standard has been rapid. Parquet evolved from collaborations between Cloudera, Twitter and other companies to provide a "columnar storage" container file format. This means data is stored on disk per column rather than per row like with Avro.

Storing data per column has many immediate efficiency benefits. The size of data on disk is greatly reducing by encoding and compressing data on columns. The data in columns typically have less variance than the data in rows, and therefore have a higher compression ratio. Data analysis is more efficient since data scans can be limited to just the columns of interest. For example, in the Avro scheme, to perform exploratory analysis on the channels ingested, our data analysis scripts had to read in the entire ingested row for each record including all metadata and the waveform blob. With Parquet, that same analysis is much faster since it only loads the columns under immediate consideration.

The initial release of Parquet was limited to Hadoop I/O classes and query tools. Since then Spark and other projects have significantly increased support for Parquet, making it the current leader in specialized Big Data file formats.

JavaScript Object Notation (JSON)

JSON (Crockford, 2009) is a widely adopted human-readable format originally developed as a communication mechanism between web servers and browsers. The format consists of key-value pairs and has support for encoding strings, numbers, arrays, and objects. Where JSON truly stands out compared to the more specialized formats discussed above is in the breadth of its adoption. Many APIs and applications include built-in support for JSON, and all major programming languages include JSON parsing libraries. Apache Spark can read and perform queries on JSON documents just as easily as it can with Parquet and AVRO files. Using JSON as a destination format makes it easy to export the metadata catalog to Oracle and use the data in other presentation layer technologies. For on-disk space considerations, easy integrations, and Spark compatibility, our ingestion and transformation layers will use JSON as the intermediate data format during processing and to store metadata details.

Code Written

A Spark job was created to take files in the data lake and package them for use in the transformation layer. The packing process consists of parsing the raw files, adding ingestion metadata, combining like objects (such as CSS site and StationXML files), and writing these model objects out to the JSON file format for further processing in the transformation layer. The main runner is the PackageRawData class, as shown in 8 below. This class first gets all the file paths under the input directory provided by the user, a necessary step because the Spark

Context wholeTextFiles method will not do this file tree recursion itself. Once we have the file paths, the contents of the files are read as a String and a JavaPairRDD is returned, where the absolute file path is the key and the file contents are the value. This paired RDD is the data lake RDD. The file processors repeatedly use the data lake RDD so it is cached/persisted to memory and disk in serialized form.

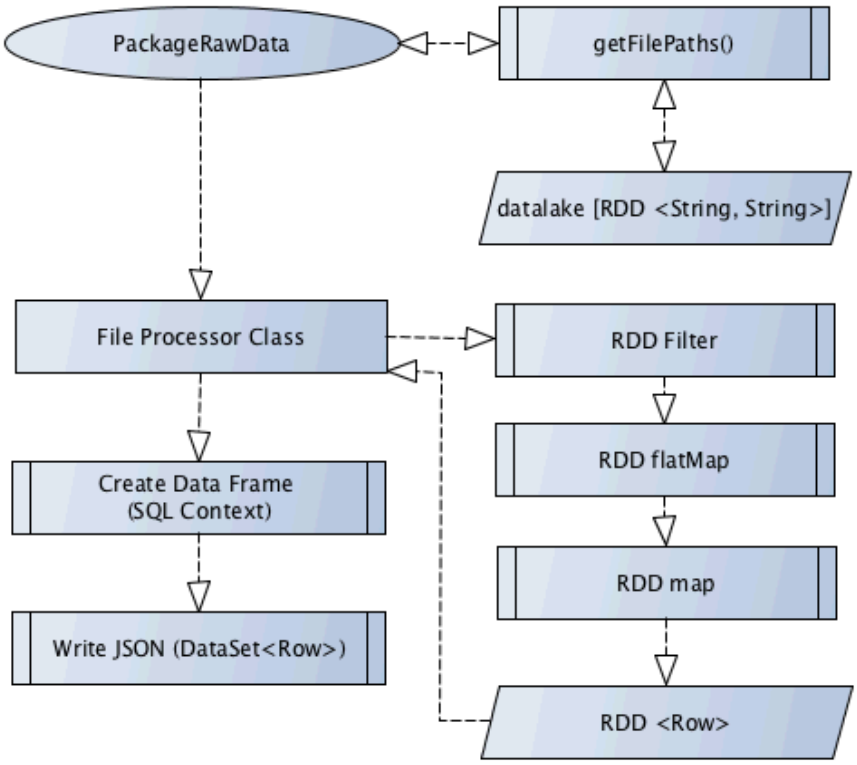

Figure 8 Program flow for turning raw files into JSON documents

Next, each of the file processors is called and they all follow similar steps. The first step is to filter the data lake RDD to only include the files that the executing class is responsible for processing. We have file processors for all the CSS and SEED types needed as input to create the unified schema. Once filtered, the RDD is put through a flat map transformation. The flat map takes each line from the file and creates an instance of one of the model classes. These model instances are then transformed by the map operation to be Row instances. This conversion to Row instances will go away in the future when the DataFrame or Dataset can be created directly from the Java object instances. Once we have the data represented as Row instances we can combine the data with a schema provided by the model class to create a DataFrame instance (Dataset<Row> in Spark 2+). Now that the Rows are a DataFrame, the DataFrame's write method is called to output the data in JSON format to HDFS. The written JSON file is now

ready to undergo the post-processing in the Transformation later required to create the unified schema.

Discussion

Many different file formats and storage solutions exist. We considered a subset of available technologies which would provide the best support for our use case. In evaluating different technologies, we looked at the ability to store the objects on different file systems (HDFS, NFS, Blob/document store), the ability to query the data, the ability to compress the data, and most importantly, the level of adoption and tooling around the standard.

We first evaluated keeping the files in their raw format, but this did not provide good I/O throughput and was incompatible with the design of HDFS which prefers a few large files over thousands of small files. Next, we evaluated the key-value NoSQL database HBase. HBase is good for storing schema-less rows that can grow to millions of columns, but we found the use of heavy Region Servers, I/O throughput, and size limitations for rows to not support our overall data storage needs, although it may be a good solution for our metadata catalog. We looked at Avro and Parquet file formats which provide self-documenting schemas inside of the output files, row and column level compression, and the ability to quickly query data. We decided against both formats as their adoption outside of big data tools is still limited. We ultimately decided that JSON provided the most flexibility and compatibility with our storage use-cases.

A Spark program was written that processes the raw seismic metadata and waveforms into a JSON representation. These JSON output documents are then consumed by the transformation, integration, and presentation layers. Additional work needs to be done to apply basic quality checks to the data before propagating to the transformation layer which will do the more extensive quality checks.

Layer 3: Transformation

The transformation layer is where incoming parsed and standardized data from the ingestion layer is transformed into curated data for the new data mart. The majority of data quality assessments and data integrations are performed in this layer. The input data for this layer is the cumulative ingested data in the unified schema format, and the output is cleaned and analyzed data for the presentation layer.

Quality Metrics

Data coming into the transformation layer has already been checked and validated against the unified schema definition in the ingestion layer, and unit conversions and other data transformations have been made as necessary. However, there are numerous metadata errors

that can survive these kinds of checks. The initial work in the transformation layer is to perform quality checks on the data in the unified schema and document the results. These metrics are valuable for an accurate understanding of the data, and they can subsequently be used by integration logic, analysis applications, and end users as appropriate.

For this project, we developed a set of metadata quality metrics to identify and document problematic data. As we worked with the data during the project we added checks for each of the metadata issues we came across. The list of 38 quality checks we developed is in Table 1. This list is by no means comprehensive, but includes many of the problems that commonly plague us in our existing archive. The checks were implemented as a series of procedures that generate an error log for each table, row, and error found. Results of the checks are discussed in the section on the presentation layer.

Table 1 Quality checks performed on the data in the unified schema

Table	Check Procedure	Error Description
network	check_parent(network,source)	parent does not exist
network	check_parent(network,source)	parent epoch does not exist
network	check_dates(network)	epoch start > end
network	check_dates(network)	epoch start = end
network	check_dates(network)	epoch overlap
network	check_dates(network)	epoch duplicate
response	check_parent(response,stream)	parent does not exist
response	check_parent(response,stream)	parent epoch does not exist
response	check_dates(response)	epoch start > end
response	check_dates(response)	epoch start = end
response	check_dates(response)	epoch overlap
response	check_dates(response)	epoch duplicate
station	check_parent(station,network)	parent does not exist
station	check_parent(station,network)	parent epoch does not exist
station	check_dates(station)	epoch start > end
station	check_dates(station)	epoch start = end
station	check_dates(station)	epoch overlap
station	check_values(station)	LAT out of range
station	check_values(station)	LON out of range
station	check_values(station)	ELEV out of range
station	check_dates(station)	epoch duplicate
station	check_values(station)	missing LAT/LON
stream	check_parent(stream,station)	parent does not exist
stream	check_parent(stream,station)	parent epoch does not exist
stream	check_dates(stream)	epoch start > end
stream	check_dates(stream)	epoch start = end

stream	check_dates(stream)	epoch overlap
stream	check_values(stream)	LAT out of range
stream	check_values(stream)	LON out of range
stream	check_values(stream)	ELEV out of range
stream	check_values(stream)	DEPTH <0
stream	check_values(stream)	DIP <> ORIENTATION_CODE
stream	check_values(stream)	SAMPRATE <=0
stream	check_values(stream)	SAMPRATE out of SEED bounds
stream	check_values(stream)	missing LAT/LON
stream	check_distance(stream)	distance to parent > .1km
stream	check_distance(stream)	elevation >1km from parent
stream	check_dates(stream)	epoch duplicate
waveform	check_parent(waveform,stream)	parent does not exist
waveform	check_parent(waveform,stream)	parent epoch does not exist
waveform	check_dates(waveform)	epoch start > end
waveform	check_dates(waveform)	epoch start = end
waveform	check_dates(waveform)	epoch overlap
waveform	check_values(waveform)	SAMPRATE <=0
waveform	check_values(waveform)	SAMPRATE out of SEED bounds
waveform	check_dates(waveform)	epoch duplicate

Data Integration

Our legacy data mart stores a version of the data that is independent of the original source of the data. In reality, there are often multiple sources for the same data, and we may extract data from each to get the most comprehensive set. Unfortunately, all sources do not always provide mutually consistent versions of the same data, and significant inaccuracy may be introduced into our curated data while trying to sort this out. The unified schema avoids this problem by keeping data by source, but our vision going into this design effort was that application code and end users would still expect source independent data.

Our original plan for the transformation layer was to integrate multi-source data wherever we could do so accurately. Data that could not be integrated with a high confidence of accuracy would be routed off to an error pool. The pool would be kept from growing though continuous oversight, periodic analysis, and improvement of the pipeline. This system would forward only integrated data to the data mart, and allow us to transition to this new pipeline without immediately breaking or changing any downstream code.

The following sections describe our progress on the implementation of this design. Because of this work, however, our plans for the scope of the transformations done in this layer have

changed. We may consider a new name (such as “Analysis”) for this layer going forward that better describes its modified function.

Input Data Preparation

To develop and test our ingestion and integration codes we used data from 13 different sources. Two of the sources provided data in CSS format and the remainder were retrieved as Station XML from data centers that support the FDSN Station web services. A summary of the input data is shown in Table 2.

Table 2 Summary of input data used for the integration test by source

SOURCE	FORMAT	NETWORKS	STATIONS	STREAMS	RESPONSES
IRISDMC	SEED	198	11,234	242,018	241,868
NCEDC	SEED	21	3,561	49,525	49,440
USNDC	CSS	2	2,038	4,939	4,822
INGV	SEED	27	718	7,928	7,928
GEOFON	SEED	30	718	8,752	8,750
RESIF	SEED	11	484	7,493	0
SED	SEED	16	474	3,691	8
UNR	CSS	3	375	5,886	5,892
ORFEUS	SEED	29	222	3,280	3,259
USPSC	SEED	2	139	1,054	1,053
IPGP	SEED	5	122	2,064	1,568
LMU	SEED	1	120	414	414
NIEP	SEED	4	102	874	874

Prior to integrating the station data, we performed several checks for internal (within source) consistency. This resulted in the removal of 514 STATION rows as shown in Table 3. Because of referential integrity constraints, removing those rows also resulted in the removal of 1,587 STREAM and 1,448 RESPONSE rows.

Table 3 Counts of STATION rows removed due to inconsistencies

Reason	Removed Rows
End time <= begin time	1
Inconsistent station positions	26
Overlapped station epochs	332
Station epoch not contained in network epoch	155

STREAM data were also subjected to some consistency checks prior to integration. In total 8,371 rows were removed, with the results summarized in Table 4. Of the rows removed for overlapped epochs, 2524 were from a single station (H20) in the H2 network and most of the

rest were from just two other networks. The bulk of the STREAM rows removed for sample rate problems were from stations of the PB (Plate Boundary Observatory) network and were for band 'Q'. The SEED manual specifies that 'Q' should have a sample rate of < 10^-6, and these channels had a claimed rate of 0.05. However, the SEED manual also says that these are approximate values, so our procedure may have been too aggressive.

Table 4 Counts of STREAM rows removed for various inconsistencies

Reason	Removed Rows
Stream end time <= begin time	12
Stream sample rate outside range for band code	3387
Overlapped epochs	4764
Orientation code inconsistent with dip	208

Responses were tested for internal consistency (non-overlapped epochs and response epochs fully contained in STREAM epochs). We found 2272 that failed the second of these conditions. We also tested the usability of the response files. The first check was a simple existence check: Does the path specified by "dir/dfile" exist? There were 3573 failures. These failures were due to the fact that integration testing was on a snapshot that referenced files in the data lake. Because more work was done in the ingestion layer after the snapshot was taken, changes were made that invalidated some database content.

We also tested response file usability by attempting to de-convolve each response from synthetic data, and found nearly 78,000 responses that failed. This is a surprisingly large number of failures. Based on examination of the exceptions and the set of channels involved, we think the root cause is that the JEvalresp code is brittle with respect to "exotic" channels and RESP files produced by certain organizations. Table 5 lists the top 12 networks with failed responses. The transportable array is the hands-down winner with almost 27,000 failures.

Table 5 The top 12 networks by total count of failed responses

CODE	Network	Failures
TA	USArray Transportable Array	26997
PB	Plate Boundary Observatory Borehole Network	8919
EM	Electromagnetic Studies of the Continents	6309
CI	Southern California Seismic Network	4458
BK	Berkeley Digital Seismic Network (BDSN)	3958
IV	Italian National Seismic Network	3083
N4	Central and Eastern US Network	2584
SN	UNR NSL Southern Great Basin	1641
GE	GEOFON Program, GFZ Potsdam, Germany	1502
IU	Global Seismograph Network (GSN - IRIS/USGS)	1202

RO	Romanian Seismic Network	1055
UL	USGS Low Frequency Geophysical Data Network	1015

Of the nearly 27,000 failures for TA, almost 23,000 are from the channels (LCQ, VEA, VPB, VEC, VKI, OCF, ACE, LOG, LKM, LDM, LIM, BDF, LDF, LDO, and BDO). None of these are conventional seismic channels. Nearly all the exceptions were in some way related to input unit specification, decimation specifications, or epoch end date specification.

Response Filtering

Although being readable and parse-able is a minimum requirement for response usability, if a response does not have the correct amplitude and phase characteristics it is worse than not being available at all. Much of the processing done by GMP seismologists requires that waveforms be corrected from raw counts to ground motion (e.g. velocity in m/s). Such corrections are generally accomplished by de-convolving the instrument response from the seismograms. If the de-convolution succeeds, but gives the wrong answer it leads to incorrect, and potentially hard to trace, research results.

As part of this integration effort we attempt to identify problematic instrument responses and exclude them from the results. For vertical-channel data available by FDSN Station Service, we can compute P-wave amplitudes for large teleseismic events and compare to the amplitudes expected based on the mb magnitude estimates from global catalogs. For each vertical-component FDSN channel with a sample rate of at least 10 Hz and for which we can retrieve waveforms via FDSN Station Service we:

- Identify up to 15 events with 5.2 <= mb <= 6.2 at distances from 15 to 40 degrees that occurred during the response epoch.
- For each of these events, we retrieve waveform data from 100s before P to 140s after P.
- We remove the response and filter using a 2-pole Butterworth filter with corners at 1 and 3 Hz.
- We then measure the zero-peak amplitude (A_{meas}) of the mean-removed absolute value of the signal from 10s before P to 40s after.

Next, we use the reported mb magnitude to predict the amplitude that should have been observed:

$$A_{pred} = T * 10^{(mb - Q(\Delta,h))}$$

Q(Δ,h) is computed using a table of Q values retrieved from (http://www.jclahr.com/science/software/magnitude/mb/qtab.txt). We then record the ratio:

$R = {A_{meas}}/{A_{pred}}$.

We evaluated 15,829 responses in this manner. The histogram of log10 amplitude ratios and a fit to a t location scale distribution are shown in Figure . The distribution parameters are (μ=.4655, σ=.3799, and ν=1.777).

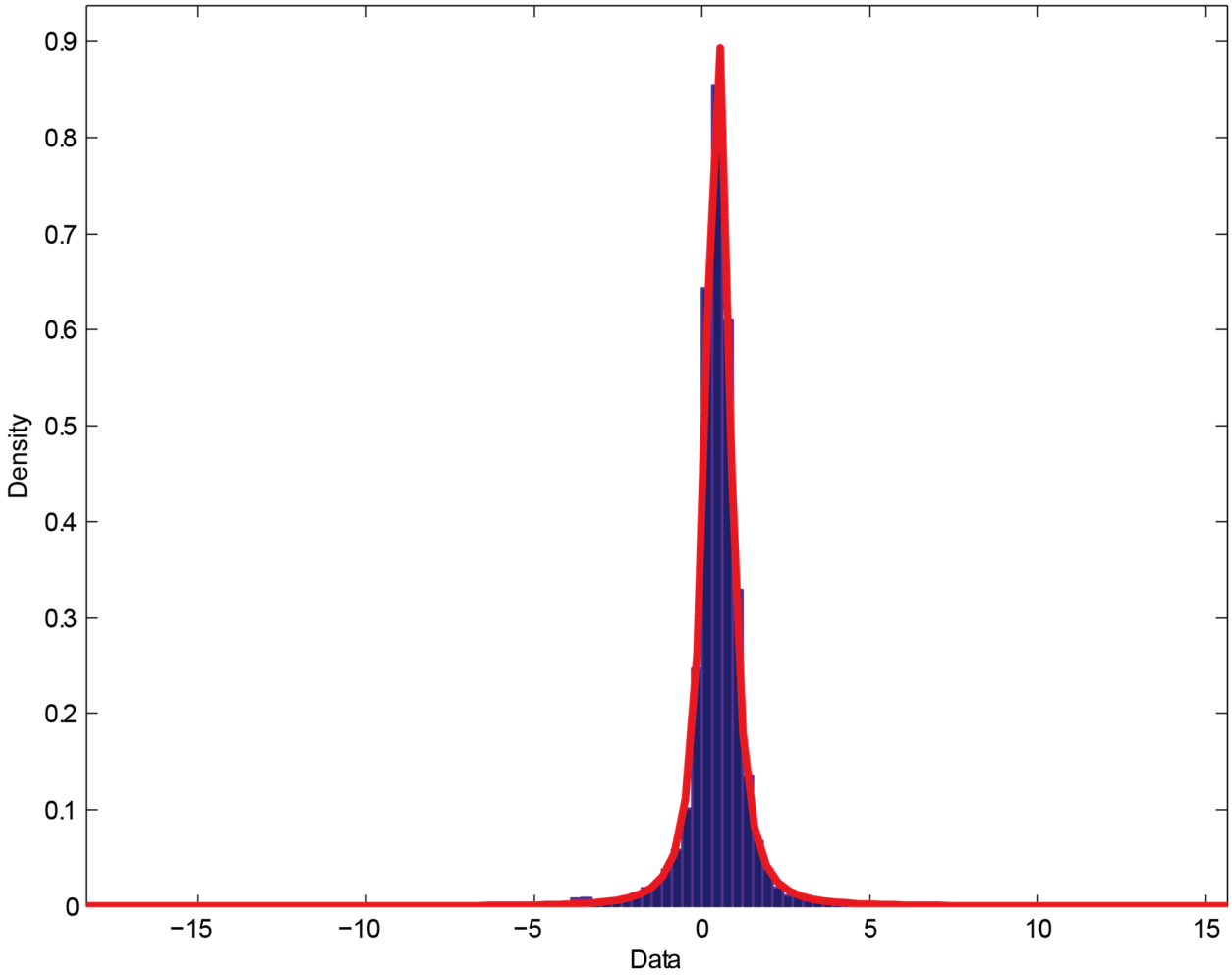

Figure 9 A histogram of 〖log〗_10 R (blue) with a t location scale distribution fit to the data (red)

Using the distribution parameters, we can identify responses that are statistical outliers using a T test. In this experiment, we eliminated 385 such responses at the 0.95 level. We also used these results to select from among multiple candidates during a multiple-source merge.

Although this experiment demonstrates the possibility of empirically evaluating response correctness, we were only able to test about 7% of the integrated responses using the mb amplitude comparison approach. Only a subset of channels can be processed this way, and we didn't have access to the necessary waveform data to support all of those.

Preliminary Integration

The tables in Figure 10 have the same structure as the input tables, except that the former alternate keys are now surrogate primary keys. Also, derivable columns have been removed from the natural keys. For example, STATION_P does not include either NETWORK_CODE or SOURCE_CODE since both are derivable through joins with NETWORK_P and SOURCE_P respectively. Finally, array-specific columns have been removed from STATION_P. Their functionality has been moved to a new set of tables dedicated to arrays.

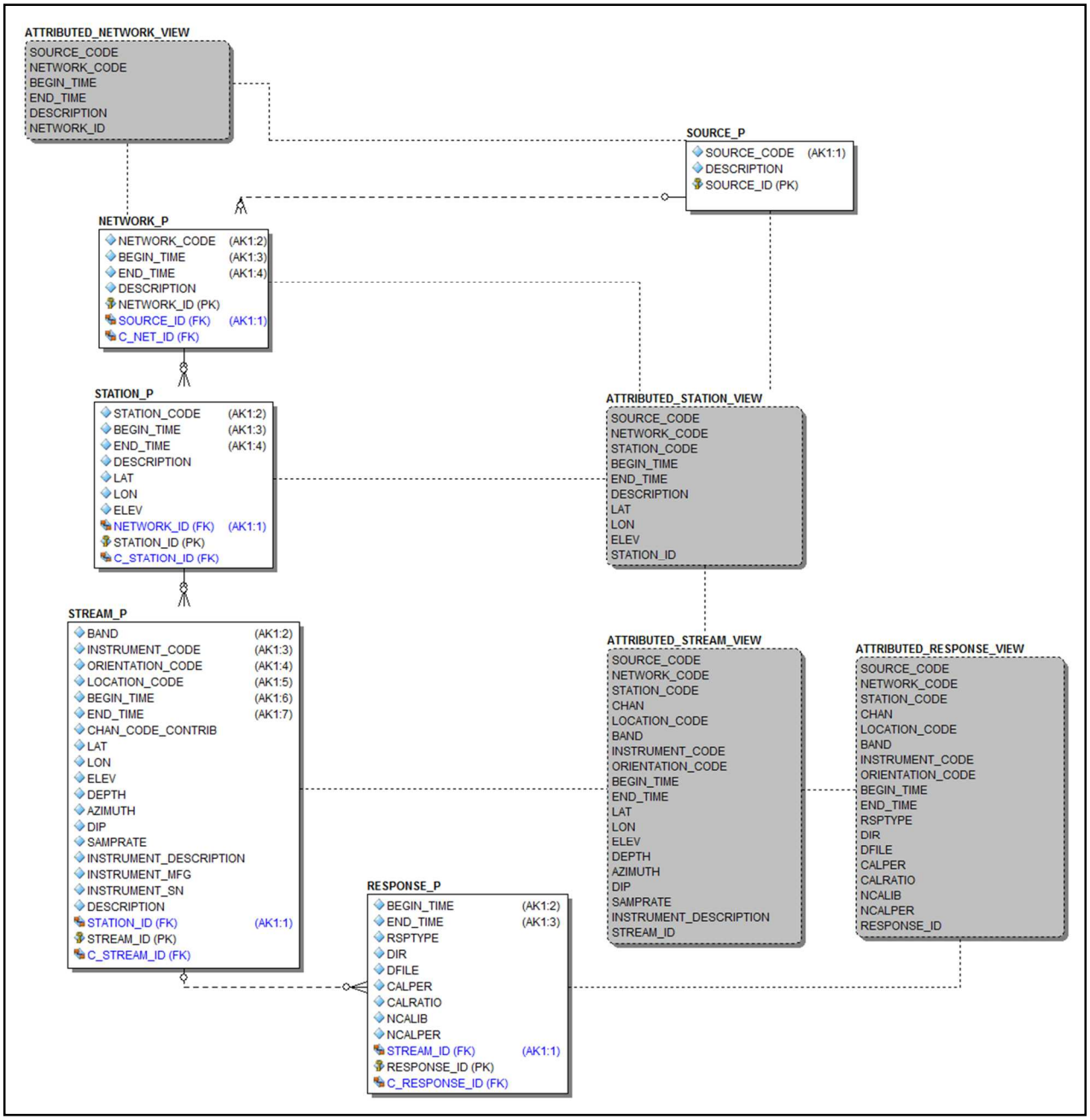

Figure 10 The tables for "cleaned" metadata into which the prepared input data was ingested

A sample of the results of this first stage of integration are shown in Table 6. Note that these data are only integrated in the sense that multiple sources have been combined into a single set of tables and the data within sources has been "cleaned" to improve usability and consistency. Table 6 shows how sources fared in producing integrated STATION_P, STREAM_P, and RESPONSE_P data. At the STATION_P level, all sources did well. On average 97% of the input data were retained overall. At the STREAM_P level the average was about 93% of rows retained. Integration of response data was disappointing for all sources. The average retention was 56%, not counting one particularly troublesome case where all responses were dropped because of epoch conflicts.

Table 6 Sample summary of the integration results for stations, streams, and responses

	Stations			Streams			Responses		
SOURCE	**In**	**Final**	**Percent**	**In**	**Final**	**Percent**	**In**	**Final**	**Percent**
IRISDMC	11234	11199	99.7	242018	236527	97.7	241868	186058	76.9
NCEDC	3561	3461	97.2	49525	47684	96.3	49440	36135	73.1
GEOFON	718	711	99.0	8752	8605	98.3	8750	3234	37
INGV	718	678	94.4	7928	7568	95.5	7928	3314	41.8
ORFEUS	222	220	99.1	3280	3248	99.0	3259	1996	61.3
IPGP	122	122	100	2064	2059	99.8	1568	1093	69.7
USPSC	139	137	98.6	1054	990	93.9	1053	849	80.6
NIEP	102	102	100	874	868	99.3	874	240	27.5
LMU	120	120	100	414	414	100	414	246	59.4
SED	474	453	95.6	3691	3183	86.2	8	3	37.5
RESIF	484	484	100	7493	7486	99.9	0	0	

Integration Between Sources

The tables shown in Figure 0 provide a means of storing data from multiple sources in a way that maintains consistency. By including network code as a key and by maintaining a consistent time representation across tables, two major sources of inconsistency are removed. If the presentation layer incorporated just those two changes then the main change to application queries would be the inclusion of NETWORK_CODE. Adding SOURCE as part of the key complicates logic a little more, and our original hope was that we could avoid this by combining data provided by multiple sources. In this section, we present some findings.

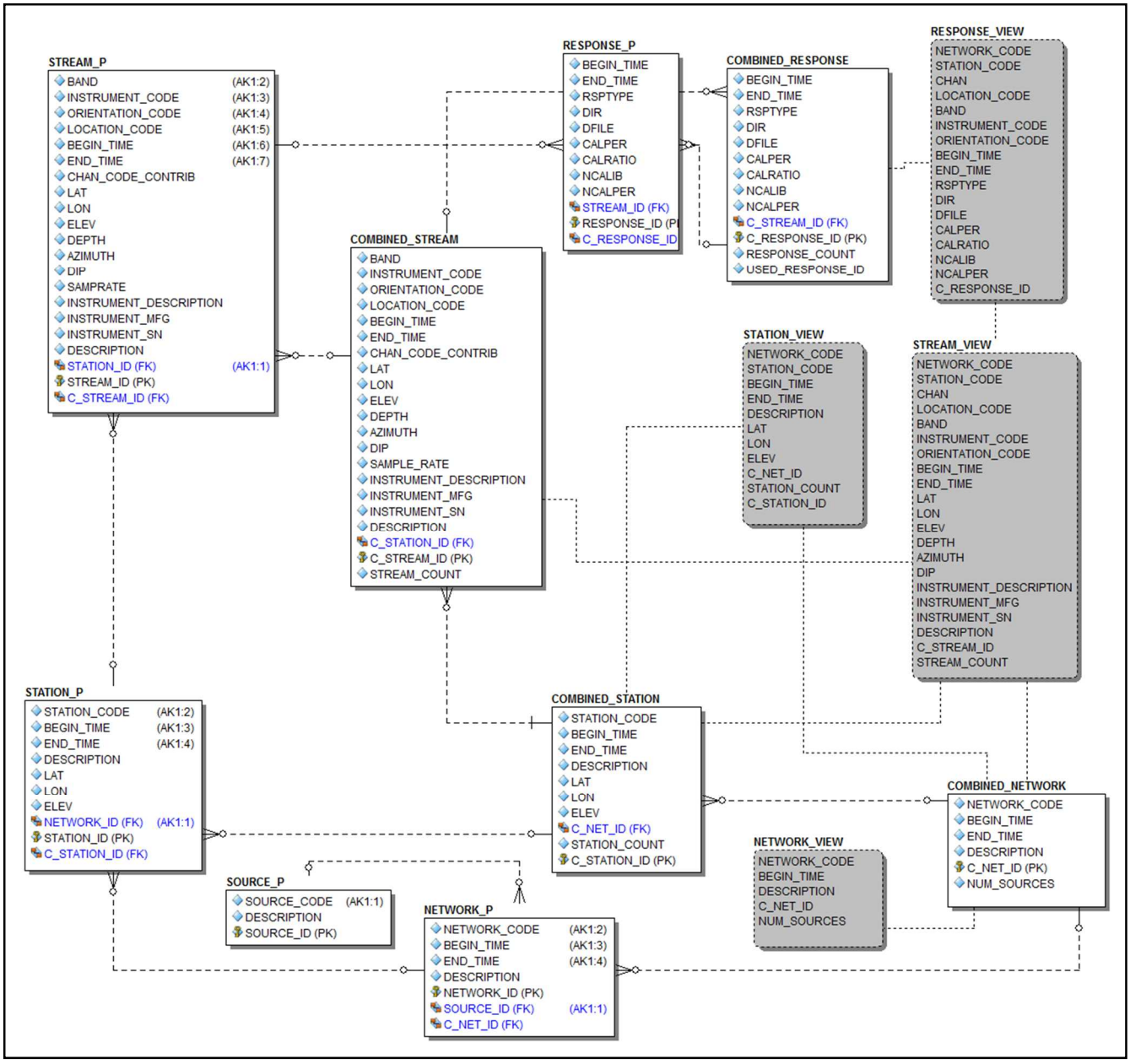

Figure 41 The schema for the source-integrated data. Tables are shown in white and views in gray. The tables with the "COMBINED" prefix hold the combined data, and the tables with the "P" suffix hold the potentially multiple rows which have been used to create a single row in the corresponding "COMBINED" table.

Figure 41 shows the schema into which the data were combined. Our strategy for combining rows was simple. Station epochs were merged if they matched to the nearest day. Merged station rows hold the averages of latitude, longitude, elevation. The begin time was set to the earliest of the epochs being merged and the end time was set to the latest end time. Any remaining input rows that overlapped merged rows were dropped. STREAM data were handled in an analogous manner except that begin times and end times were set to the average of the

input times. Responses were handled a little differently. The time resolution used was one second and instead of averaging fact columns, we chose a single response. If each response had an entry in the amplitude ratio table, we chose the one with the smallest deviation from the mean. Otherwise, we chose the first response.

A summary of the merged data is shown in Table 7. The counts shown are for distinct rows disregarding time. Time-sensitive row counts differ by less than 3%. Some data loss has occurred in merging between sources, but overall, the results seem encouraging.

Table 7 The results of the two merge strategies.

Data Type		**Attributed Data Retained**		**Combined Data Retained**	
	Raw Totals	**Surviving**	**Percent**	**Surviving**	**Percent**
Station: Distinct NET-STA	17,344	17,121	99%	17,121	99%
Stream: Distinct NET-STA-CHAN-LOCID	185,576	181,537	98%	180,534	97%
Response: Distinct NET-STA-CHAN-LOCID	181,997	125,499	68%	124,526	68%

Although we have shown that large fractions of the metadata may be combined successfully, it is still not clear that this is the right thing to do. Combining the data is a lossy operation. Once combined, whether by the algorithms discussed above, or by some alternative; unless the inputs are identical, the data appear to be more certain than they really are. Perhaps more importantly, when we combine waveform data based on keys like SOURCE or NETWORK, we don't really know without checking the counts whether the data are identical or not. The following examples illustrate some of the issues.

First, we consider integrating data having the same station code and position, but with differing network codes. After preliminary integration, we found 376 station codes that are associated with more than one network code. We identified 620 pairings where a single station code had two or more rows with identical positions, overlapped epochs, and with two different network codes. The optimistic assumption (with respect to merging waveforms) is that in these cases the same physical station has been reported by different networks, so that in processing seismograms we can ignore the source.

We have not retrieved waveform data for these pairings, so we don't know whether seismograms would match count-for-count, but we do have the responses. If the responses are identical then it is reasonable to expect that waveforms would match as well, assuming sample rates are the same. There are 5694 response pairings associated with the station pairings. For each pairing, we de-convolved each instrument response from an impulse function. The results are shown in Figure 52(a) and (b).

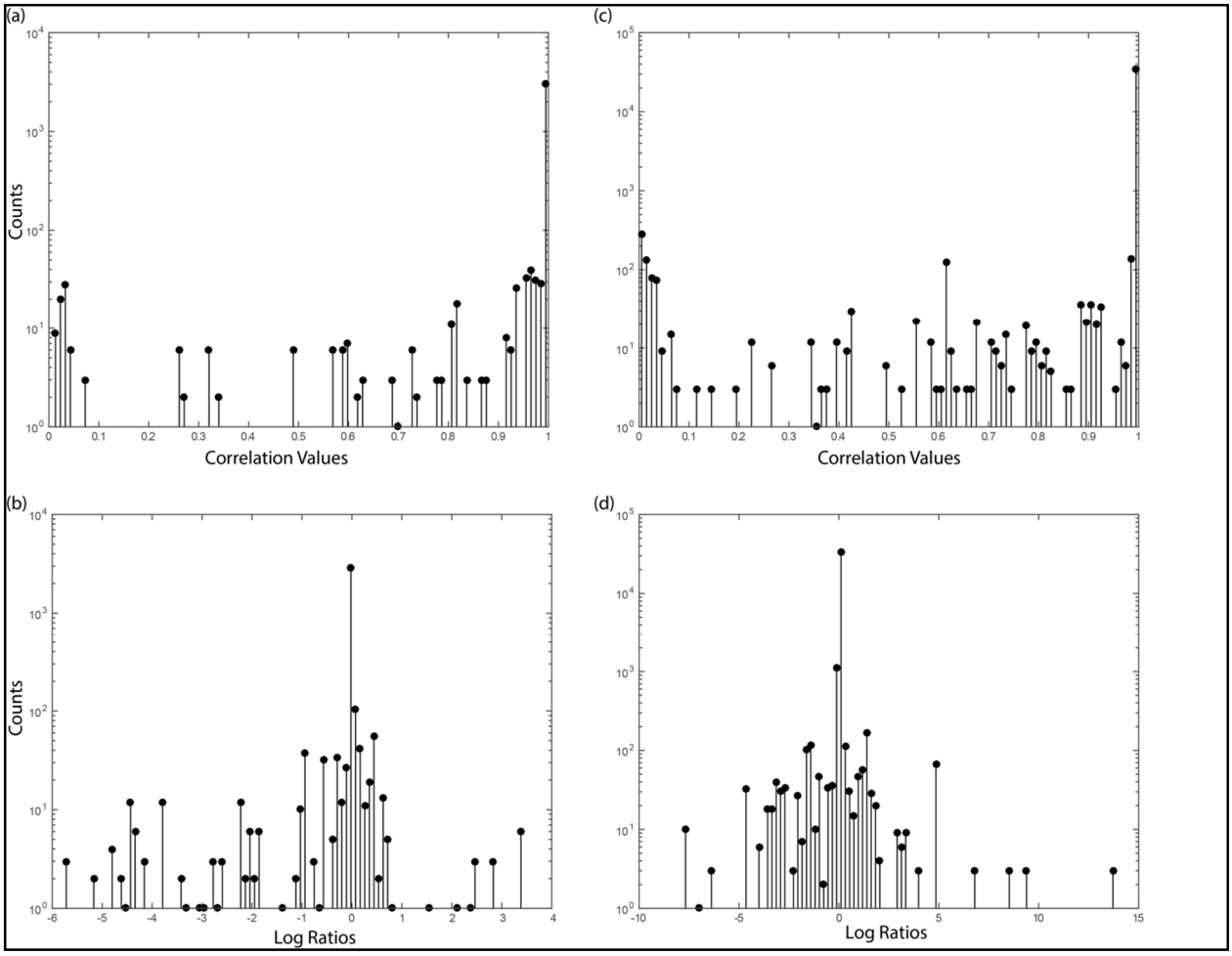

Figure 52 A comparison of 5694 instrument responses in terms of the cross correlation of de-convolved impulse functions (a) and ratios of peak amplitudes of the de-convolved signals. In each case the responses being compared have the same station code, channel name, location code, and time period. They differ in network code.

Panel (a) shows binned correlation values of de-convolved impulse functions and (b) shows the ratios of peak amplitudes of the de-convolved signals. In the great majority of cases, the de-convolved signals match. However, there are hundreds of instances where the signals are poorly correlated, the amplitudes don't match, or both. Clearly the responses are different in these cases, but without further investigation we cannot be sure whether this is strictly a response problem, or whether the responses differ because the data streams differ. Either way, it does not seem advisable to merge these streams until the issues are better understood.

What about merging data when all keys match except for source? We already know that data from the US NDC may be inconsistent when merged with data from IRIS because the NDC data has scaling information held with the waveforms while IRIS holds that information in the response files. But can we merge FDSN data from different sources? In our test data set, there are 1116 instances of the same (FDSN) net-station-epoch provided by more than one source.

There are 35,848 corresponding response pairs, and their binned comparisons are shown in parts (c) and (d) of Figure 52. Although most responses appear to be identical, there are hundreds that do not match. It is unclear if the mismatches are due to errors, or because the data have been treated differently.

Discussion

We experimented with two approaches to integrating metadata (network-station-stream-response) from 13 different sources. In the first approach, we kept the data separated by source and removed data found to be unusable, or that violated one of several checks for correctness. Using this approach, about 99% of station data, 98% of stream data, and 68% of response data were successfully integrated. The second approach built on those results by merging data from different sources where other keys matched. Nearly all data survived the final merge step, so that the final percentages changed little from those of the first merge (Table 6).

In discussing those results, we noted that the final merge step introduced ambiguity because we could not be sure that stream data from different sources would be identical even if the keys matched. We showed that response data provided by different sources does not always match. This could imply differences in waveforms as well, and suggests that at least until we understand the differences in responses, it may not be advisable to merge data from different sources.

Although our integration strategies succeeded in producing a self-consistent metadata collection that retained a large fraction of the input metadata, metadata important to researchers was screened out. For example, all UNR responses were removed because of epoch inconsistencies. Thousands of stream rows (and hence responses) were removed for epoch inconsistencies and other rule violations.

The problem we are facing is that these schemas reflect an idealized world in which perfect records are always kept. In that world, you can depend on response epochs being subsets of stream epochs, which are themselves subsets of station epochs, and so on. Data that conform to those expectations can be successfully integrated and are usable without any anomalies. The remainder either must be dropped or somehow modified to be consistent.

Any modifications to the data to make them consistent are necessarily arbitrary and merely provide the illusion of consistent and certain information. Instead, if we want to use all the metadata that comes our way, the only practical approach is to employ a schema that allows the inconsistencies and to accept that some queries will produce ambiguous, multi-valued results.

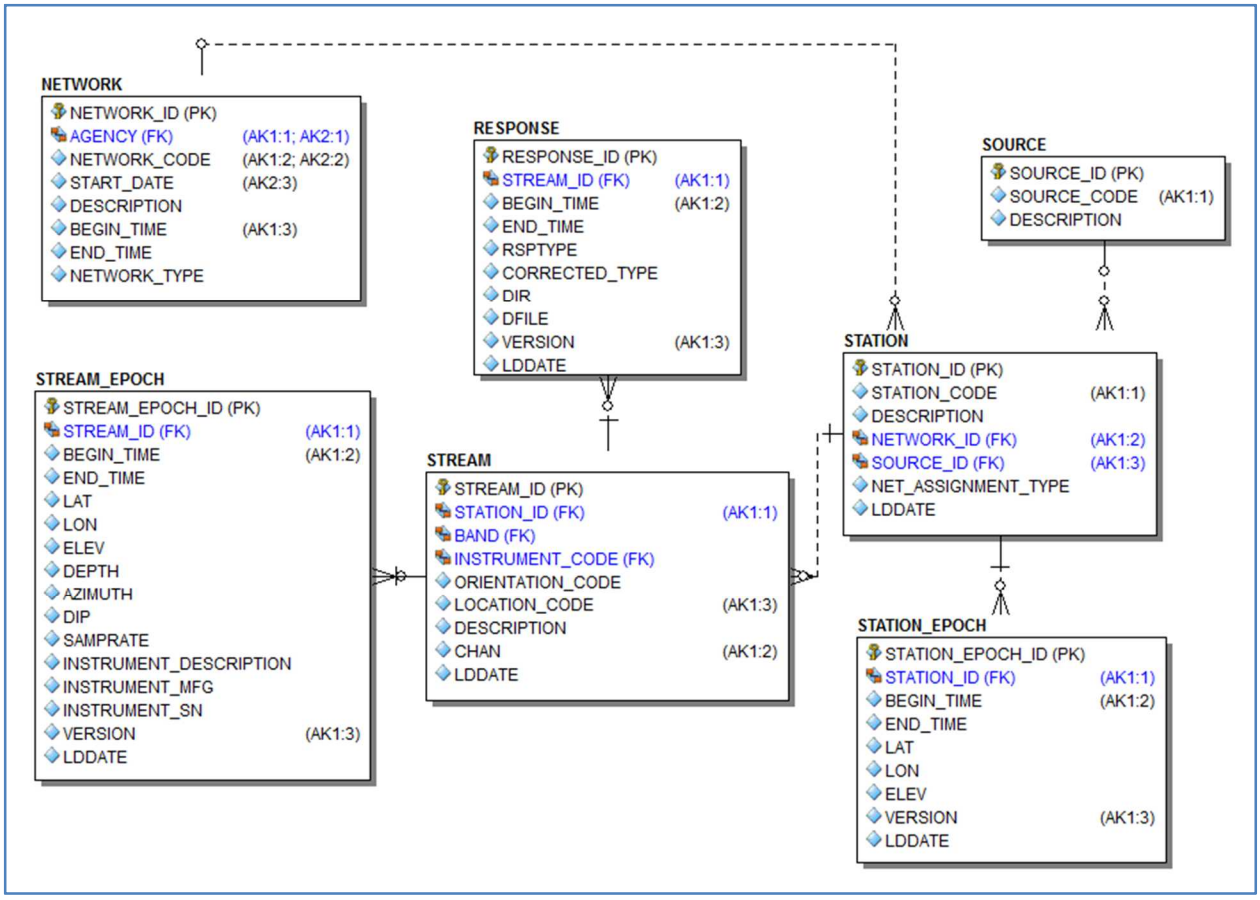

Figure 63 A possible schema that allows for inconsistent epochs.

One approach that maintains the source-network-station-stream hierarchy is to simply pull the epoch-sensitive information from station, and stream (Figure 63). In this design, it is still true, for example, that a stream belongs to a station. But, there is no requirement that any stream epoch will belong to a given station epoch

As an example, suppose we need the instrument response for a waveform. To satisfy this, we join WAVEFORM to RESPONSE on STREAM_ID subject to the WAVEFORM times being contained in the RESPONSE times. This will return 0 to N rows. Of course, we would like a single row, so an algorithm will be required to down select. Similar strategies apply for STREAM information and STATION information. With a design like this, we can load all the metadata we ingest. We can still check for rule violations, e.g. invalid STREAM sample rates, etc. But instead of dropping the rows, we can flag them. Of course, there is a cost. Making this work would require a major reworking of our data processing infrastructure. It will also become more difficult for researchers to perform ad hoc queries, since they will need to accommodate multi-valued results.

Layer 4: Presentation

The presentation layer is where fully processed data is made available to end users. It is the external interface to the data mart and may include multiple views of the curated data tailored to different use cases. Ultimately it will replace our current production schema for our downstream analysis applications and researchers.

Our initial implementation plan for this layer was to create views of the processed and integrated data from the new pipeline that closely emulated the structure of the existing data mart. This strategy promised big benefits. First, it would allow us to do a direct comparison of the results with the existing system to validate the new pipeline. We could run the two systems side by side and compare results over time until the new system was proven. Once that was accomplished, it would provide a mechanism for seamlessly moving users and applications to the new system with very little interruption. Unfortunately, the issues described in the last section show that we cannot reliably integrate multi-source data into a source independent schema like the one in our existing system.

Confidence Measures

The presentation layer can also include any reports, dashboards, or summaries of the data that could be useful to data consumers. In the new data mart users and applications might have to select the best option from multiple rows, where they used to always get one. A measure of confidence in each row could be particularly valuable in helping them decide. Ideally it would be a single comparison metric that incorporated the quality, verifiability, and stability aspects of the data.

As a first pass at this idea we created a summary quality metric based on a penalty value associated with each error logged during the quality checks in the transformation layer. The total penalty of errors not found is divided by the total possible penalty to give a measure of "goodness" of the metadata row. The design allows penalties to be weighted to show relative importance, but for this exercise we accrued the same penalty for each error.

Even this overly simplistic algorithm provided some insights into the data, and supported quick quality comparisons between similar data from different sources. It also identified problem areas and error trends in the data that require further probing and analysis. One of the summary quality reports we developed for the presentation layer is shown in Table 8.

910 **Table 8 Sample summary of data quality by data source**

SOURCE	TABLE	ROW COUNT	GOOD ROWS	GOOD%	PROB ROWS	PROB%	ERROR COUNT	AVG CONF	MIN CONF
GEOFON	network	30	30	100%	0	0%		1	1
GEOFON	station	718	718	100%	0	0%		1	1
GEOFON	stream	8752	8737	100%	15	0%	15	1	0.94
GEOFON	response	8750	8750	100%	0	0%		1	1
GEOFON	TOTAL	18250	18235	100%	15	0%	15	1.00	0.94
INGV	network	27	27	100%	0	0%		1	1
INGV	station	718	680	95%	38	5%	38	0.99	0.9
INGV	stream	7928	7339	93%	589	7%	610	1	0.88
INGV	response	7928	7924	100%	4	0%	4	1	0.83
INGV	TOTAL	16601	15970	96%	631	4%	652	1.00	0.83
IPGP	network	5	5	100%	0	0%		1	1
IPGP	station	122	122	100%	0	0%		1	1
IPGP	stream	2064	1936	94%	128	6%	128	1	0.94
IPGP	response	1568	1568	100%	0	0%		1	1
IPGP	TOTAL	3759	3631	97%	128	3%	128	1.00	0.94
IRISDMC	network	198	198	100%	0	0%		1	1
IRISDMC	station	11234	11196	100%	38	0%	38	1	0.9
IRISDMC	stream	242018	222305	92%	19713	8%	20412	0.99	0.81
IRISDMC	response	241868	238807	99%	3061	1%	3061	1	0.83
IRISDMC	TOTAL	495318	472506	95%	22812	5%	23511	1.00	0.81
NCEDC	network	21	21	100%	0	0%		1	1
NCEDC	station	3561	3462	97%	99	3%	99	1	0.9
NCEDC	stream	49525	43920	89%	5605	11%	6498	0.99	0.75
NCEDC	response	49440	49440	100%	0	0%		1	1
NCEDC	TOTAL	102547	96843	94%	5704	6%	6597	1.00	0.75
NIEP	network	4	4	100%	0	0%		1	1
NIEP	station	102	102	100%	0	0%		1	1
NIEP	stream	874	868	99%	6	1%	6	1	0.94
NIEP	response	874	874	100%	0	0%		1	1
NIEP	TOTAL	1854	1848	100%	6	0%	6	1.00	0.94
ORFEUS	network	29	29	100%	0	0%		1	1
ORFEUS	station	222	220	99%	2	1%	2	1	0.9
ORFEUS	stream	3280	3148	96%	132	4%	138	1	0.88
ORFEUS	response	3259	3259	100%	0	0%		1	1
ORFEUS	TOTAL	6790	6656	98%	134	2%	140	1.00	0.88
RESIF	network	11	11	100%	0	0%		1	1
RESIF	station	484	484	100%	0	0%		1	1
RESIF	stream	7493	7007	94%	486	6%	492	1	0.88

RESIF	TOTAL	7988	7502	94%	486	6%	492	1.00	0.88
SED	network	16	16	100%	0	0%		1	1
SED	station	474	451	95%	23	5%	23	1	0.9
SED	stream	3691	2960	80%	731	20%	783	0.99	0.81
SED	response	8	8	100%	0	0%		1	1
SED	TOTAL	4189	3435	82%	754	18%	806	0.99	0.81
USPSC	network	2	2	100%	0	0%		1	1
USPSC	station	139	139	100%	0	0%		1	1
USPSC	stream	1054	1021	97%	33	3%	33	1	0.94
USPSC	response	1053	1053	100%	0	0%		1	1
USPSC	TOTAL	2248	2215	99%	33	1%	33	1.00	0.94

Arrays

As mentioned in the unified schema description, the STATION table has the columns STATYPE, REFSTA, DNORTH, and DEAST which are taken from the CSS SITE table. These columns are populated for the subset of stations that are part of an array. For most stations, the columns are unset. The inclusion of these columns in STATION is problematic for several reasons. The first is that nearly all station rows have these four columns which carry no information about the station. STATION is being used to describe three different kinds of entity, (seismic station, array element, seismic array). As such, it violates the 1NF requirement of having a separate table for each set of related data.

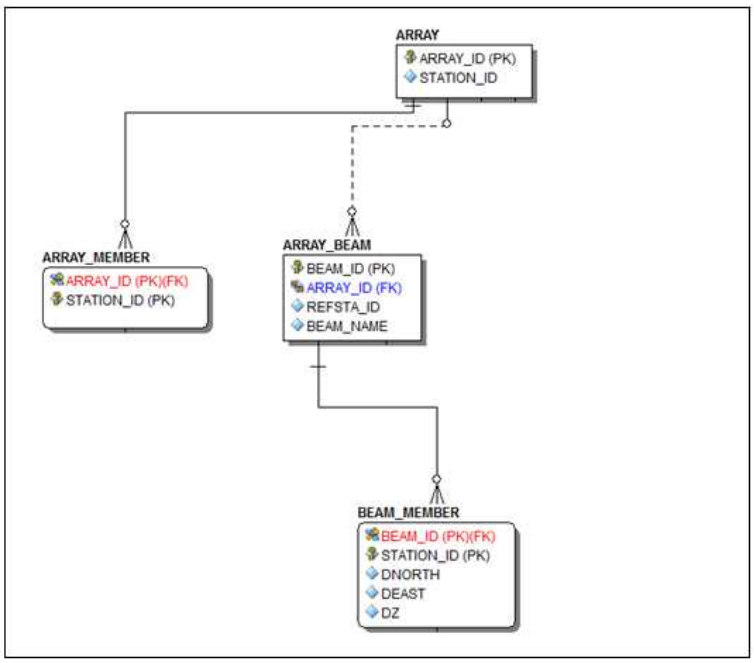

Figure 14 The tables used to describe arrays and array beams

A more flexible way to describe seismic arrays and beams is shown in Figure 14. In this schema, the ARRAY table contains a row for every array epoch. Every STATION row with STATYPE value of 'ar' that survived integration is referenced in this table by its STATION_ID. The ARRAY_ID is a surrogate key generated from a sequence. The ARRAY_MEMBER table was populated from surviving STATION rows with STATYPE = 'ss' and REFSTA matching an array row on STATION_CODE and epoch.

Station Clusters

Unlike our existing system, the new pipeline makes no attempt to map data for the same physical station to the same station code. Not only do different data sources use different identifiers for the same station or channels, but sometimes even the same source changes the codes due to input error, changes in naming conventions, data corrections, etc. A significant problem is also created if the same identifier is used for different physical stations.

Matching location coordinates often cannot be used to sort out these issues. For example, stations may be physically moved over time, and array members may be very close together. We know from our current system that changing station identifiers to make a global naming standard where one doesn't exist just creates error and uncertainty. The new pipeline does not try to correct or assign names, but this is far from perfect because the end user is left to identify and fix any naming problems in the data they use.

Stations clusters are an attempt to provide the end user with information about other stations that may be the same as the one they are interested in. It is implemented as a procedure that walks through the station table and makes location based clusters of stations based on proximity. If the current station being considered is within 0.1 KM of any station already in a group, the station is added to the group. Once all stations are processed any groups containing a common member are coalesced, and a unique integer is assigned to each group for identification. Array members are disregarded.

Using this logic 15,845 distinct station codes are grouped into 14,810 clusters, indicating that there are possibly 1,035 superfluous station codes in the data. The maximum distance between any two stations in the same cluster is less than .33 km. Table 17 is a summary of the top clusters with the most members, and it illustrates some of the additional insight station clusters give into the data.

The first row of Table 17 shows a cluster with 70 different station codes, all from the same source and at identical locations. All 70 stations are in the GY network from the IRISDMC source. None have a station name or description. Most likely this is a data error of a new type that would not be caught by the quality checks in the ingestion or transformation layers.

957 **Table 9 Top station clusters with the most members**

SCLUSTER_ID	SOURCE_CNT	NET_CNT	STA_CNT	MIN_DIST	AVG_DIST	MAX_DIST
16258	1	1	70	0	0	0
15901	2	2	37	0	0.01	0.088
15728	2	2	23	0	0	0
17003	1	1	14	0	0	0
19213	1	1	12	0	0.098	0.319
16039	1	1	12	0	0.007	0.024
16150	2	2	11	0	0	0
19177	2	3	11	0	0.033	0.162
16607	1	2	11	0	0.034	0.08
16117	1	1	10	0	0.037	0.104
16124	2	3	9	0	0.044	0.107
18130	2	1	9	0	0.041	0.089
16811	1	1	9	0	0.03	0.075
18290	1	3	8	0	0.056	0.125
19204	2	5	7	0	0.081	0.205
16101	1	1	7	0	0.107	0.322
15935	1	1	6	0	0.016	0.039
16203	2	2	6	0	0.053	0.149
16247	1	1	6	0	0	0
16643	1	2	6	0	0.058	0.157
24014	1	1	6	0	0.021	0.057
17190	1	2	6	0	0.032	0.084
18281	2	2	6	0	0.009	0.049
22162	2	2	6	0	0	0
17100	1	1	6	0	0	0
15940	2	2	5	0	0.002	0.007
16710	1	1	5	0	0.063	0.149
18111	2	3	5	0	0.026	0.06
18193	2	2	5	0	0.001	0.011
18243	2	3	5	0	0.033	0.082
25616	1	1	5	0	0.046	0.11
18286	1	1	5	0	0.011	0.044
18375	1	1	5	0	0.026	0.093
19885	2	2	5	0	0.002	0.008
22896	2	1	5	0	0	0
22936	2	6	5	0	0.054	0.141
18253	1	1	5	0	0.033	0.069
16051	2	2	4	0	0.01	0.035
16516	1	1	4	0	0	0

A search of the data for all clusters from one source and network with multiple stations at the exact same location shows that there are 162 similar clusters. The great majority of these (152) have only two stations, and except for the 70-station cluster already mentioned, the rest have less than fifteen members each. A legitimate reason for this phenomenon is multiple stations at the same site but with different elevations or different types of sensors (hydro-acoustic, infrasound, etc.). Other examples appear to be array members that are not identified on the array list, and are mistakenly identified by the same coordinates. In fact, many of the clusters on the list in Table 17 appear to be array members that are not on the array list. This is likely because they are from SEED sources, and the array list is constructed solely from CSS sources.

Station clusters appear to be a useful way to identify related stations, arrays, and some new types of data errors. The distance of 0.1 km was somewhat arbitrary, and further analysis may indicate a more useful value. A conservative distance limit risks eliminating related stations that are just outside of the limit, but a generous limit may return false positives. A different clustering technique, such as machine learning, may give better results by using all the station metadata to group related stations instead of only the location coordinates. Station codes, descriptions, and elevations provide valuable insight and could be used to form and then classify clusters as data errors, arrays, legitimate location changes, naming differences between sources, etc.

Conclusion and Future Work

There is significant work left to do before GMP has an operational data pipeline with the new design. Many ideas for follow-on and remaining work are discussed in the previous sections of this report. All our implementations were prototypes, which allowed us to avoid some critical decisions to meet scope and funding constraints. For example, no final hardware or environment decisions were made. Multiple copies of data were kept at every processing stage for convenience, with no regard to storage limitations. A complete data lifecycle and archiving policy needs to be established. Applications will need to be modified, and users will need to be trained to use the new data mart. New user interfaces and applications should be developed to expose and use the quality metrics, data provenance, and other new data elements we created.

Despite these limitations this project was an important start toward the development of a next generation ingestion pipeline for GMP. We validated NiFi as an overall flow manager, created a unified schema to store data from all anticipated formats, implemented basic functionality in each layer of the new architecture, verified that it solved many of our existing ingestion issues, and uncovered and quantified many of the errors and deficiencies in seismic data. We also greatly increased our understanding of the issues and our expertise in scalable data management, which has already led to other funded projects.

References

Anderson, J., W.E. Farrell, K. Garcia, J. Given, H. Swanger (1990). "CENTER FOR SEISMIC STUDIES VERSION 3 DATABASE: SCHEMA REFERENCE MANUAL". Science Applications International Corp. Center for Seismic Studies 1300 N. 17th Street, #1450 Arlington, VA 22209-3871.

Anderson, Q. (2013) "Storm Real-Time Processing Cookbook", Packt Publishing, Birmingham, U.K. ISBN: 1782164421 9781782164425

Bridgwater, A. (2015). "NSA 'NiFi' Big Data Automation Project Out In The Open". *Forbes (magazine)*. Retrieved 2016-09-21.

Crockford, D. (2009). "Introducing JSON". json.org. Retrieved July 3, 2009.

George, L. (2011) "HBase: The Definitive Guide: Random Access to Your Planet-Size Data", O'Reilly Media Inc., Sebastopol, CA. ISBN-13: 978-1449396107.

Islan, M. K. and A. Srinivasan (2015) "Apache Oozie; The workflow scheduler for Hadoop", O'Reilly Media Inc., Sebastopol, CA. ISBN-13: 978-1449369927

Norris, M.W.; Faichney, A.K., eds. (2002). *SEG Y rev1 Data Exchange format* (PDF). Tulsa, OK: Society of Exploration Geophysicists

NSA (National Security Administration) press release, 2014. "NSA Releases First in Series of Software Products to Open Source Community". https://www.nsa.gov/news-features/press-room/press-releases/2014/nifi-announcement.shtml

About the FDSN. (n.d.). Retrieved March 22, 2017, from http://www.fdsn.org/about/

IRIS (Incorporated Research Institutions for Seismology), 2012. SEED Reference Manual (Standard for the Exchange of Earthquake Data), SEED Format Version 2.4, August, 2012. https://www.fdsn.org/seed_manual/SEEDManual_V2.4.pdf

Jain, A. (2013) "Instant Apache Sqoop", Packt Publishing, Birmingham, U.K. ISBN: 1782165762 9781782165767

Primack, D. "LinkedIn engineers spin out to launch 'Kafka' startup Confluent". *fortune.com*. Retrieved February 10, 2015.

Romanowicz, B. (1990). "The Federation of Digital Broad Band Seismic Networks", Laboratoire de Sismologie, Institut de Physique du Globe. Retrieved from: http://www.fdsn.org/media/_s/publications/historical/fdsn_report_romanowicz.pdf.

Russel, J. and R. Cohn (2012) “Apache Avro”, Book on Demand, ISBN 5511968579, 9785511968575.

Vance, A. (2009). "Hadoop, a Free Software Program, Finds Uses Beyond Search". *The New York Times*. Archived from the original on August 30, 2011. Retrieved 2010-01-20.

White, T. (2015) “Hadoop: The Definitive Guide 4th Edition”; Chapter 13. O’Reilly Media Inc., Sebastopol, CA, ISBN: 9781491901687.

Zaharia, M. et al (2016). Apache Spark: A Unified Engine for Big Data Processing, *Communications of the ACM*, 59(11):56-65.

Appendix

A Short Word on Waveforms

Most this paper has been focused on parametric data, the primary focus of our efforts to date and the main source of our ingestion problems historically. Presently our seismic data is largely made up of segmented files of varying length stored in simple file structures on a large NFS file storage array indexed by SQL tables. This has historically proven adequate but looking at the problems we are facing with our parametric data, the increasing rate of ingestion overall, and increased demand from our research staff, we have been thinking about potential alternative solutions.

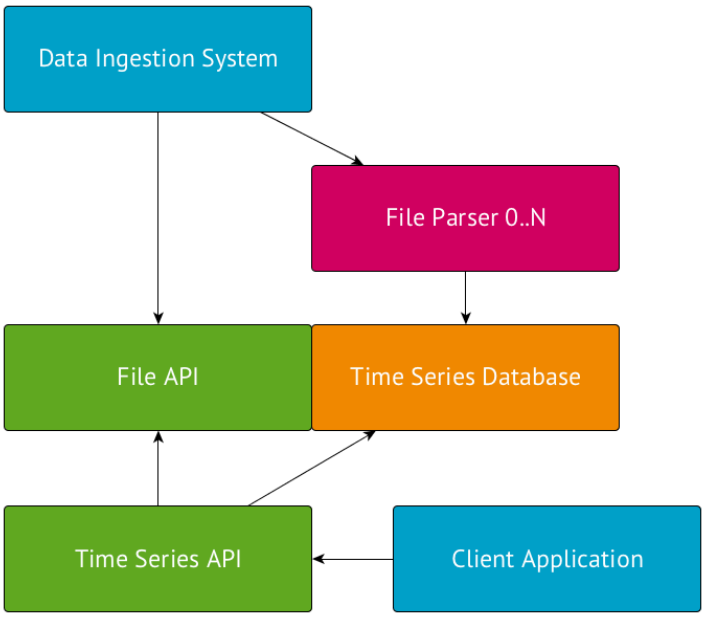

Figure 7 An abstraction of a waveform ingestion pipeline

Figure 15 is a potential solution. In this diagram, the data ingestion system stores a raw copy of the waveform as a file object in its original format in the data lake. It also passes a copy of the data to a file processing sub-system that transforms the raw waveform into a series of key-value metadata tuples that are stored in a time-series database.

The time-series database can then act as a hot cache for the data in the data lake. The time-series API abstracts both the database and the storage infrastructure from client applications. The API service is responsible for queueing and processing “cold” data from the data lake into the hot cache as needed to satisfy incoming requests. This allows the most used data to remain resident in a fast and consistent format that client applications can easily query. It also allows

for some level of abstraction around segmentation since client applications can simply ask for ranges of time and the time-series service is responsible for assembling the segment on the back end.

Considerable thought will need to be put into how the data is organized and keyed in the data lake to avoid access contention ("hot-spotting") and to leverage distributed storage as efficiently as possible. Likewise, the metadata tagging and keying in the time-series database needs careful consideration to avoid negative performance and scaling impacts from over-duplication of data, skewed keys, and poor partitioning. Metadata storage will also need to be altered based on the selected database implementation to ensure that any queries on the metadata itself can be served efficiently. Our time-series database needs to support differing time scales per series and potentially within the series; something many of the current offerings are not designed to handle. Work has only just begun on this and there are already many issues to be resolved.

An Aside on Software Infrastructure

As part of our effort to redesign and re-implement our ingestion pipeline we evaluated our software development and deployment methods to look for process improvement opportunities. Our legacy ingestion tools typically tend to make very specific assumptions about the configuration and existence of infrastructure such as databases, libraries, and other dependencies.

Virtually all our software and infrastructure above the basic operating system level is also hand installed and configured. This is acceptable while we only have a handful of servers to maintain but it makes the system brittle to change and limits our ability to scale or make other meaningful upgrades very quickly.

With the goals of increasing our flexibility, facilitating software deployments, and improving scalability moving forward, we have started to rethink our development and deployment tool chains. To achieve our stated goals our software environments must be both portable and reproducible. Given those requirements we are starting to migrate our software infrastructure to container based deployment schemes and are beginning work on implementing orchestration tooling at every layer of the stack.

The ingestion pipeline provides an example of this strategy. A software developer should be able to run a single script (Figure 16) on his local machine and get a single node version of the complete software environment for the pipeline running locally in a virtual machine. This allows for the entire system to be deployed or re-deployed simply by running orchestration tools against existing physical or virtual hosts with the container runtime installed.

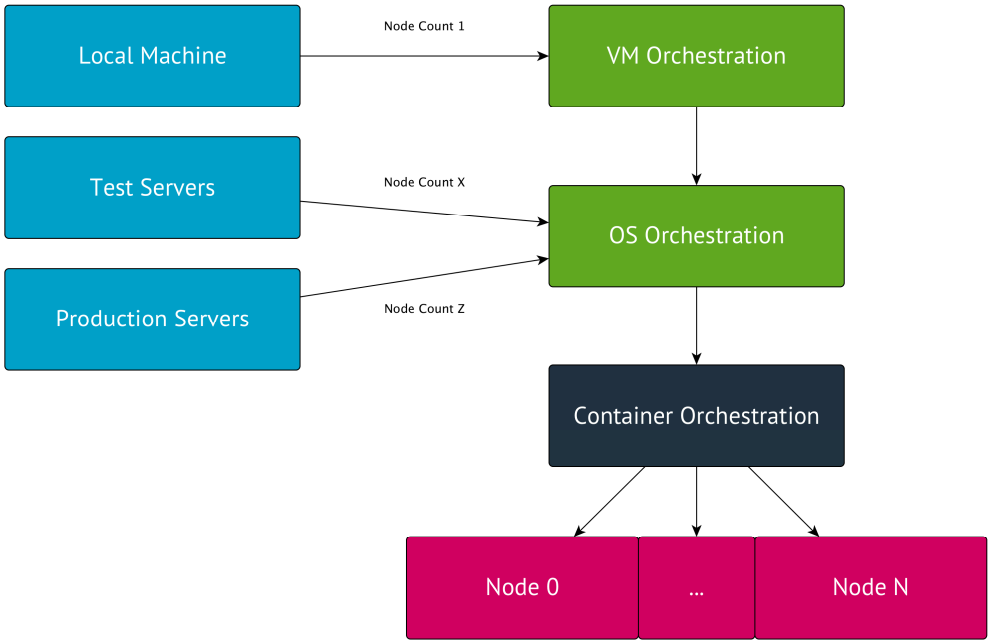

Figure 8 Orchestration stack

The fully contained developer environment is configured and running exactly as it would be on test and production servers, just at a greatly reduced scale. Once the developer is confident in their changes they need only move the container with their components to the test server and then on to deployment. Migration in this way is a simple scaling operation and allows for tighter integration of automated tools for deployments and testing.

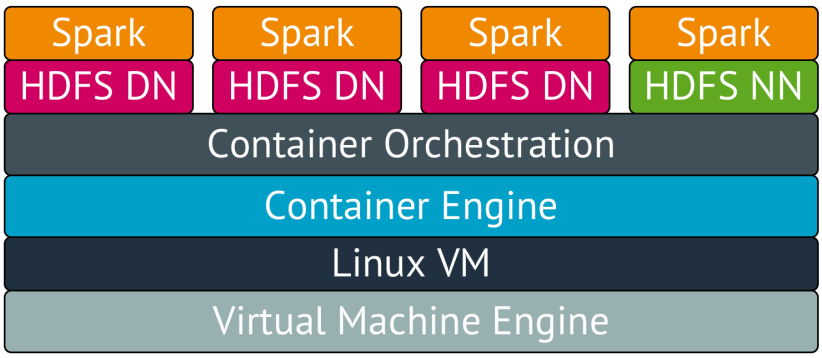

Figure 9 A single node deployment

Figure 17 shows a single node Spark cluster with eight containers running on the node: three Spark worker containers, one Spark master, three HDFS data-nodes, and a HDFS name-node. While this is a greatly simplified version of a software stack, it allows a developer to test their algorithms on their local machine while still exercising all the same code execution behaviors that will be seen on a production cluster. For a developer to move their code to a real cluster they only need to change the location pointer for the Spark master.