Handling Detector Characterization Data (Metadata) in XENONnT

Luis Sanchez[1,*], *Yossi* Mosbacher[2,**], and *Aaron* Higuera[1,***] *Christopher* Tunnell[1]

[1]Department of Physics and Astronomy, Rice University, Houston, TX 77005, USA
[2]Department of Particle Physics and Astrophysics, Weizmann Institute of Science, Rehovot 7610001, Israel

Abstract. Effective metadata management is a consistent challenge faced by many scientific experiments. These challenges are magnified by the evolving needs of the experiment, the intricacies of seamlessly integrating a new system with existing analytical frameworks, and the crucial mandate to maintain database integrity. In this work we present the various challenges faced by experiments that produce a large amount of metadata and describe the solution used by the XENON experiment for metadata management.

1 Introduction

Many experiments collect a large amount of very homogeneous primary data, usually in the form of massive collections of time series. The challenges associated with these large time series are mostly centered around the sheer scale of the data and the complexity associated with partitioning and versioning such large datasets while ensuring data integrity. While these primary datasets comprise the overwhelming majority of data by size, these experiments tend to produce a large number of smaller auxiliary datasets. These auxiliary datasets (metadata), sometimes referred to as conditions data[1], track auxiliary elements like the detector conditions, data correction values, manual operations, and other auxiliary measurements, each with their own schema and time resolution. The main challenges of managing metadata comes mostly from the heterogeneity of these datasets and their usage.

Experiments, especially those with large datasets, need 2 key components: a place to store the data and a way for scientists to access this data, [2, 3]. Particle physics experiments such as those at the Large Hadron Collider (LHC) have a rich history when it comes to metadata management [4]. As such, most of the literature in the field is in the context of collider experiments. Despite the difference in experimental setup, we face similar challenges when it comes to metadata management. The ATLAS collaboration, for instance, wrote about some of the challenges they face with conditions data, its management, and its interface [5]. Here we will iterate on some of these issues, express the requirements we need for a metadata management system, discuss our implementation of a system to manage our metadata, and discuss its integration with the current software used in the XENON experiment for data processing.

---

[*]e-mail: las19@rice.edu
[**]e-mail: joe.mosbacher@gmail.com
[***]e-mail: ahiguera@rice.edu

2 Background

2.1 Metadata characteristics

We will start by listing some of the main characteristics of our metadata so the reader can better judge whether our experience and solutions are relevant to them.

1. Heterogeneity: Metadata can span from simple information, like timestamps and run numbers, to more complex calibration and environmental data, including temperature, pressure, and electromagnetic conditions. Standardizing and managing diverse types of metadata is a significant challenge.

2. Multiple Sampling Frequencies: Metadata can be sampled with frequencies ranging from seconds to months.

3. Immutability: Most of our metadata is used to produce scientific results and therefore is immutable to ensure reproducible outputs. This does not mean that updates are not made, just that those updates are stored as new versions instead of replacing the existing values.

2.2 Requirements on metadata management

2.2.1 Single Source of Truth

Experimental metadata used for processing scientific results is required to be consistent and immutable in order to achieve reproducible results. There are two common patterns when it comes to the storage of versioned data: the centralized "single source of truth" and the distributed git-like approach. Both have various advantages and disadvantages, but in the case of metadata used for processing, it is necessary to have a single data repository that is used for all "production" processing.

2.2.2 Development Environments

It is important to understand that the processing chain is a collection of complex software components that are under constant development. Given that the metadata is used in the processing chain, it is effectively part of this development process and therefore needs to accommodate small development environments that can be used easily during development. The flexibility of a distributed system is necessary for analysts doing the preliminary analysis. The preliminary corrections and quality cuts are used in producing the metadata in the first place. Detector condition corrections, for example, can be highly inter-dependent, requiring a flexible and rapid development cycle. Such a rapid development cycle would be highly impaired by a centralized immutable database taking part of the development process. It is therefore crucial to have the ability to set up small, local staging databases for analysts to use during development and the review process.

2.2.3 Multiple Data Access Patterns

There is a large variety of access patterns required for scientific metadata. Many researchers from various institutions often collaborate on projects sharing the same data. The data needs to be accessed by analysts from around the world as well as from within high-performance computing (HPC) environments when processing experimental data. The metadata should be easily accessible to all collaborators regardless of their physical location, requiring a well-designed database and network infrastructure. Here again, a hybrid approach must be taken to allow for access over https via multiple REST APIs when connectivity is limited. This approach also enables direct connections via binary protocol to allow for high throughput when needed.

2.2.4 Versioning Support

A large part of experimental metadata is used in the processing of the experimental data. Some examples include data selection parameters, data correction parameters, various tuned settings of the processing algorithms, and even machine learning models. These values need to be immutable for analysis results to be reproducibility, but they also need to be versioned so that they can be tuned as the analysis improves.

2.2.5 Queries and Indexing

Most experiments produce a large variety of datasets, each with unique indexing requirements. Each dataset needs to be indexed by a unique key to support versioning. The unique key can be a composite of multiple fields. For numerical values, their dependence on the index can be either discrete or continuous. For continuous fields, there are two common ways of describing the dependent variable: step-wise (i.e. interval of validity) and sampled. The step-wise description is efficient for storing fields with a very low sampling rate or non-numerical values and sampling is more suitable for numerical metadata with a higher rate of change that can be approximated via sampling and interpolation. As the amount of stored data grows, efficient querying mechanisms become vital. It should be easy for a researcher to efficiently extract relevant subsets of data for analysis based on specific criteria.

2.2.6 User Interfaces

Providing intuitive user interfaces for researchers to input, retrieve, and analyze metadata is crucial for effective data management.

2.2.7 Access Controls

Given the importance and sensitivity of the data, robust access control measures are required to prevent unauthorized access and potential tampering as well as accidental data deletion and alterations.

2.2.8 Future-Proof Storage and Flexible Schemas

As the understanding of the detectors and their backgrounds improve, analysis techniques evolve. Metadata must be stored in a manner that is flexible enough to accommodate these evolving techniques. Furthermore, some data might need to be re-analyzed in the future as detection techniques improve or new theories emerge. Ensuring the long-term preservation of data and metadata in a format that remains accessible and understandable is crucial.

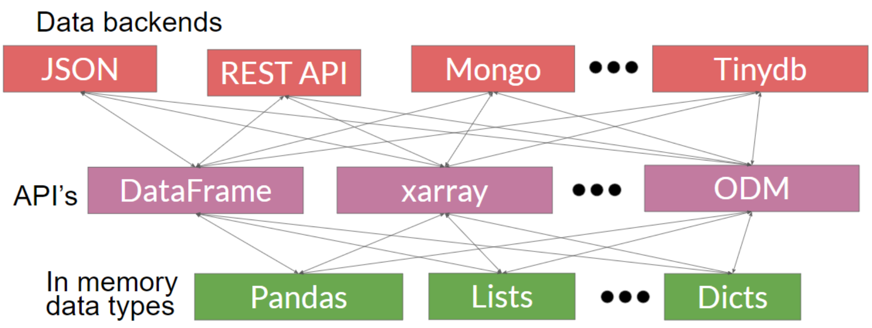

Figure 1. Diagram showing some of the data backends, APIs and in-memory data types supported by `rframe`. Each backend can use any of the available APIs and each API can handle all supported data types and vice versa.

3 Implementation

Previously, the XENON collaboration developed a simple conditions database, the corrections management tool (CMT) [6], which consisted of a collection of tables, all indexed by time with a column added for each version. These tables were stored in a central MongoDB instance which acted as the single-source-of-truth for our metadata. The advantage of such a system is in its simplicity, ease of implementation, and compatibility with pandas DataFrame structures that all analysts are familiar with. Unfortunately, our processing pipeline became very complex and required additional complexity to be added to our metadata management. The main limitations of such a straightforward approach were inefficient queries, the centralization of the data management, and uniform indexing.

`rframe` [7] is a Python package developed in XENON to satisfy the more complex requirements that arose with the complexity of our processing pipeline. `rframe` provides unified data selection APIs, `pydantic`-based definition and schema validation, and multiple data storage interfaces. The package has built-in support for multiple discrete and continuous data values and indexes as well as being extensible for custom data types. The package also provides support to assign arbitrary rules for inserting, editing, and deleting data which can be defined on a per dataset basis by simply implementing them as methods on the schema definition class. At the heart of `rframe` is the schema definition class which expands on the `pydantic` BaseModel and includes three main components: index fields, value fields and validation rules. The index fields define how data selections on stored values are converted to in-memory data, value fields define the name, types, and constraints on the values, and validation rules include arbitrary code that can be used to set complex validation rules on data before it is inserted, changed, or deleted from the database.

`xedocs` was built on top of the `rframe` framework as a separate codebase to address specific needs of the XENON collaborations. It consists of a collection of schemas for the metadata stored by the experiment as well as helper functions to set up various access patterns to the data. These access patterns include direct database access, a REST API, a Github API, and local files. To facilitate easy schema definitions, we have a collection of base classes where, at the highest level, we implemented the basic rules (such as forbidding the overwriting of data) that gets inherited by all the child classes. We then extend this collection with more specific rules for various categories of data. This allows us to minimize the amount of code needed for each schema, as most of our metadata can be inherited from one of the base classes as described in Fig. 2 and have most if not all the required functionality. Overall these classes cover most of our use cases:

- VersionedXeDoc(XeDoc): Imposes a version index.
- BaseCorrectionSchema(VersionedXeDoc): Ensures that the alias is always unique and disallows changing already set values.
- TimeIntervalCorrection(BaseCorrectionSchema): Adds an Interval index of type datetime and adds rules on which data can be appended to the database.
- TimeSampledCorrection(BaseCorrectionSchema): Adds time index of type datetime that can interpolate data and enforces rules one when these corrections can be updated.
- CorrectionReference(TimeIntervalCorrection): Base class for documents that are references to files or other documents in other storage systems.
- BaseResourceReference(TimeIntervalCorrection):Each document defines a reference to a file in our file database (internally referred to as the "resource" database).
- BaseMap(BaseResourceReference): A base class for a very common correction data structure that contains the definition of an interpolating-map instance used in our processing pipeline.

The schemas used directly for the corrections metadata consist of the descendants of BaseCorrectionSchema. The parents of BaseCorrectionSchema are used in `xedocs` to handle other metadata aspects that do not need such strict requirements. If more complex classes need to be made, these can be written directly when making a schema for the new correction, or we can make a new class that other schemas can inherit from with the desired properties. This makes the system very flexible and allows us to take full advantage of Python's object-oriented features when defining data schemas.

One example of how we use the flexibility of insertion rules as arbitrary Python code is in the treatment of what we call ONLINE and OFFLINE versions. These versions have different rules as to when each can be updated as they each serve different purposes. The ONLINE versions are used as a quick way to process our data on the data acquisition system (DAQ) in semi-realtime as data is collected. This online processing, while not very precise, allows us to quickly identify issues that can only be detected by looking at high-level data such as reconstructed energy deposition events in the detector. As such, ONLINE versions of interpolated datasets are extrapolated from their last known values. This means that as far as data immutability rules are concerned, the values from that last timestamp in the database are always set until the current time and cannot be modified. OFFLINE versions on the other hand are not extrapolated and so are only set until the last timestamp. Such a complex set of insertion rules is straightforward to implement using the `rframe` framework since it supports arbitrary pre-validation of inserts and updates using Python methods.

Furthermore, the rate-of-change for our time-dependent corrections varies greatly from correction to correction. As such, most corrections metadata are separated into time-sampled corrections and time interval corrections. The difference between these two conditions is whether we want to capture the time dependence as a series of time-sampled values that are interpolated when queried or to define a piece-wise interval-of-validity for each value. An example of a correction suitable for time-sampling is the measure of the electron lifetime [8]. Since the XENONnT experiment relies on the electrons produced by interactions with the liquid xenon atoms to reconstruct the energy of the interaction, we need to keep track of the purity of our liquid xenon, as more impurities could result in more electrons being absorbed before they produce an observable signal. We know that the number of these impurities should have a relatively continuous time evolution between short time periods. As such, it

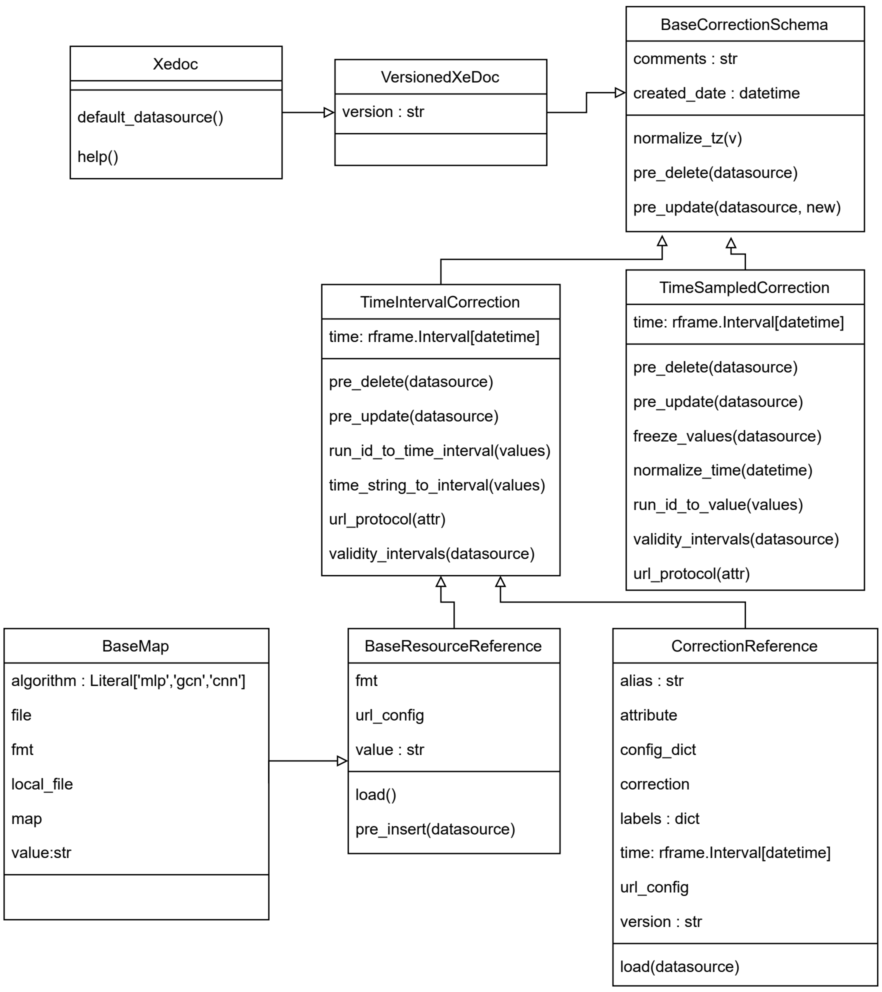

Figure 2. Diagram showing the class structure for the conditions data in `xedocs`. All conditions that are needed for event reconstruction are based on the time interval and the time-sampled schemas or their child classes. Other metadata that does not need the strict requirements of these are derived from the parent classes of base corrections but we will not be discussing these in this paper.

is best to sample the values at a high enough frequency and interpolate the values between measurements. On the other hand, we have conditions metadata such as the positions from machine-learning position reconstruction algorithms [9], which are valid for longer periods of time as the detector conditions to not change much. These cannot and should not be interpolated in time, so we simply assign to them an interval of validity for the time range on which they were validated by the analyst that created them.

4 Integration

4.1 Choice of storage

For our primary source of truth storage we chose to use our existing MongoDB replica-set with 3 replicas in Italy, Chicago and Texas for high-availability. A significant advantage of using MongoDB is its optimization for performance on reads. Our metadata is mostly immutable so it is only written once but it can be read at a very high rate due to our use of massively parallel processing on high-throughput HPC systems like Open Science Grid. For our staging databases we use a git repository with a centralized fork hosted on Github. The Github ecosystem is used mostly for its excellent review tools and automation via Github Actions. The git repository holds the code used by analysts to produce the data they insert into the database and also the data itself as json/parquet files.

4.2 Review process for data insertion

Direct insertion of data into the database is avoided as much as possible. When new data becomes available, it is first added to the git repository in a new branch and a pull request is opened on Github. When a pull request is created, an automated test runs on the proposed data to ensure it follows all the rules for the dataset it is trying to append data to, including schema and insertion rules. Beyond the automated tests, reviewers can access the data directly from Github or their local repository for testing processing with the new data. After a pull request is reviewed and authorized, it is merged into the master branch. Merging into the master branch triggers an automated synchronization script that inserts the new data into the central MongoDB database, in this insertion stage each document is validated once more against the insertion rules defined on the schema class prior to insertion.

4.3 Data access from processing workflows

Integration of `xedocs` into our current analysis software required very little modification of our analysis software straxen [10], built using the `strax` framework, a framework also built in xenon for stream processing. Thanks to its configuration management system which allows the use of abstract "URLs" that are evaluated at runtime and can run arbitrary code to retrieve the configuration values. The configuration system was simply extended to recognize URLs that start with the protocol "xedocs://" to run a `xedocs` query and fetch the configuration specified in the URL path selecting based on the URL arguments.

The specific version or correction we use for a specific analysis can change over time, either due to an improvement in the calculation of the values of these conditions or due to errors in previous conditions. To track the global configuration used for our entire processing pipeline, we also have a metadata collection that holds the various versions of each conditions dataset used for a given "global version" of the entire pipeline. These global versions consist of a collection of URLs that contain a fixed version of all conditions in order to make our analysis reproducible. When loading our analysis framework, one needs only to specify a global version and this will automatically set all the versions to use from each correction. As such, with very few modifications to the existing codebase, we were able to integrate this new system into our analysis software.

5 Discussion

The choice of which metadata management system to use for an experiment will mainly fall on the needs of said experiment. The level of complexity of the metadata and the size of the collaboration will likely be the most important factors. Here we discussed CMT, a relatively simple metadata management system, which is ideal for small collaborations with simple metadata management requirements. `xedocs` on the other hand was our solution to address the growing complexity of our metadata structure, offering many features and safeguards that could be ideal for larger experiments.

6 Conclusion

The change to using `xedocs` for our conditions metadata has greatly removed the amount of manual labor needed to maintain the conditions database. The increased flexibility of each schema allows them to be changed without having to change the structure of existing metadata. Furthermore, the restrictions of when a given collection can be updated and how it can be updated, helps preserve the integrity of the system, and creating new conditions is very easy. Furthermore, the original analysis framework does not need to be changed to accommodate changes in the metadata. Given the successful application of `rframe` in XENON, other experiments could take advantage of the existing structure in `rframe` to build their own conditions database system.

References

[1] I. Bird, Annual Review of Nuclear and Particle Science **61**, 99 (2011)

[2] J. Shiers, Nuclear Physics B-Proceedings Supplements **150**, 312 (2004)

[3] M. Jurić, J. Kantor, K.T. Lim, R.H. Lupton, G. Dubois-Felsmann, T. Jenness, T.S. Axelrod, J. Aleksić, R.A. Allsman, Y. AlSayyad et al., *The lsst data management system* (2015), 1512.07914

[4] P. Laycock, M. Bracko, M. Clemencic, D. Dykstra, A. Formica, G. Govi, M. Jouvin, D. Lange, L. Wood, *Hep software foundation community white paper working group – conditions data* (2019), 1901.05429

[5] J. Fulachier, J. Odier, F. Lambert, on behalf of the ATLAS Collaboration, Journal of Physics: Conference Series **898**, 062001 (2017)

[6] X. Collaboration, *Corrections management tool (cmt)*, https://straxen.readthedocs.io/en/latest/cmt.html (2022)

[7] J. Mosbacher, *rFrame*, https://github.com/jmosbacher/rframe (2023), version 0.2.19

[8] E. Aprile, K. Abe, F. Agostini, S.A. Maouloud, L. Althueser, B. Andrieu, E. Angelino, J. Angevaare, V. Antochi, D.A. Martin et al., Physical Review Letters **129** (2022)

[9] C. Peters, A. Higuera, S. Liang, V. Roy, W.U. Bajwa, H. Shatkay, C.D. Tunnell, *A method for quantifying position reconstruction uncertainty in astroparticle physics using bayesian networks* (2022), 2205.10305

[10] J.R. Angevaare, J. Aalbers, D. Wenz, E. Shockley, P. Gaemers, D. Xu, A. Higuera, G. Volta, Y. Mosbacher, C. Tunnell et al., *Xenonnt/straxen: v2.1.1* (2023), https://doi.org/10.5281/zenodo.8122941