IceCube File Catalog

Vladimir Brik[1], *Patrick* Meade[1,A], *Gonzalo* Merino[1], *Jan* Oertlin[1], *David* Schultz[1], and *Heath* Skarlupka[1]

[1]University of Wisconsin-Madison, WIPAC, 222 W Washington Ave. Suite 500, Madison, WI 53703

Abstract. IceCube is a cubic kilometer neutrino detector located at the south pole. Metadata for files in IceCube have traditionally been handled on an application by application basis, with no user-facing access. There has been no unified view of data files, and users often query the filesystem to locate files. Recently effort has been put into creating a unified view in a central metadata catalog. Going for a simple solution, we created a user-facing REST API backed by a NoSQL database. All major data producers add their metadata to this central catalog. Schema generation is identified as an important aspect of multi-application metadata services.

1 Introduction

The IceCube detector [1] is located at the geographic South Pole and was completed at the end of 2010. It consists of 5160 optical sensors buried between 1450 and 2450 meters below the surface of the South Pole ice sheet and is designed to detect interactions of neutrinos of astrophysical origin. An online processing and filtering service (PnF) runs in the IceCube Laboratory (ICL) at the South Pole, producing data products from the detector. After these data products are sent north to the Data Warehouse at the University of Wisconsin-Madison (UW-Madison), further analysis refine the existing data products and produce further data products (Level 2, Level 3, etc.) which are stored in the Data Warehouse. Simulation production also creates data products that are stored in the Data Warehouse.

Data products have metadata associated and stored with them in the Data Warehouse. This configuration makes it easy to identify a given data product. However, it does not solve the problem of querying for data products that match given criteria. Some examples might be "Where are all the data products derived from data taken on November 9th, 2016?" or "Where are all the data products associated with named astrophysical event X?" Because the metadata are archived with the data, performing such a query directly requires processing on the same scale as processing the data itself. Running a job on a compute cluster simply to locate data products is suboptimal.

The IceCube File Catalog (File Catalog) is an attempt to place the metadata into a single queryable service across all data applications within IceCube. The service is intended to allow interested parties to find the data they need and save the compute cluster jobs for processing the data.

A Corresponding author: patrick.meade@icecube.wisc.edu

2 Methods

One of the goals of the File Catalog was to make integration with existing applications as simple as possible. Expecting busy scientists to make extensive modifications to their software would all but ensure no adoption of the metadata service. To meet this goal, the metadata required a simple format with broad support across many programming languages. The choice of JavaScript Object Notation [2] (JSON) was natural, as the format is very simple and support libraries are available for nearly all programming languages [3].

Likewise, the application programming interface (API) to the metadata service had to be simple with broad support across many programming languages. The choice of a Representational State Transfer [4] (REST) API was natural, as the architecture is well understood and support libraries are available for nearly all programming languages. Integration then boils down to converting metadata to JSON and making a call to a REST API; a task requiring a non-trivial but tractable amount of programming effort, even for a busy scientist.

2.1 Service backend

MongoDB [5] was selected as the data store for the File Catalog project. MongoDB does not have the transactional semantics of relational databases, but does have atomic updates for individual documents. The tasks that we identified for file metadata documents fit the capabilities offered by MongoDB with respect to atomic updates for documents.

MongoDB documents are also represented as JSON. This makes a natural match between our data store and our chosen metadata format (JSON). The data store offers features to index and query JSON documents. Most of the “heavy lifting” with respect to storing and handling the data is provided by the data store itself. Our REST API interface need only leverage the functionality provided.

2.2 Service interface

The REST API was designed to provide Create, Read, Update, and Delete (CRUD) methods for file metadata documents. We define six routes in our REST API, two that deal with catalog-wide concerns, and four that deal with document-specific concerns:

- GET /api/files — Query the file catalog for metadata records
- POST /api/files — Add a new file metadata record to the catalog

- DELETE /api/files/{UUID} — Delete the metadata record identified by UUID
- GET /api/files/{UUID} — Read the metadata record identified by UUID
- PATCH /api/files/{UUID} — Update the metadata record identified by UUID
- PUT /api/files/{UUID} — Replace the metadata record identified by UUID

Each route implements some business logic to ensure some minimal conditions are met for metadata documents. Metadata documents are required to conform to a simple schema, including having a valid universally unique identifier [6] (UUID). The PATCH and PUT routes do have some functional overlap, but both are provided for the convenience of those creating and maintaining metadata documents.

2.3 Metadata document schema

Metadata documents are required to have a minimal set of metadata fields. This requirement is enforced by the logic of the RESTful routes.

- uuid — Universally unique identifier (UUID); unique
- logical_name — A name for the file; non-unique
- locations — Storage known to contain the file data; an array
  - site — Name of a storage site (i.e. DESY, NERSC, WIPAC)
  - path — The full absolute path to the file
- file_size — The size of the file in bytes
- checksum — Checksum values for the file; an object

The checksum object has algorithm names as keys (i.e. crc32, md5, sha512) and hex-encoded checksums as values. The route logic requires that file records include a sha512 checksum. Older records that lack sha512 checksums were imported with a single static sha512 checksum. These older records that still require an updated checksum can be found by querying for the static placeholder checksum. Note that the format allows for adding more checksums to metadata records in the future.

Some additional fields are not required, but are often added by convention:

- start_datetime — Start timestamp for the data in the file
- end_datetime — End timestamp for the data in the file
- run_number — Detector data taking Run Number
- subrun_number — Detector data taking Subrun Number
- first_event — First event contained in the data
- last_event — Last event contained in the data

These additional fields enable some queries, but are not required. Some types of analysis, for example a statistical analysis over many detector runs, may not have a single identifiable run number because the data are taken in aggregate. However, where these values are available, application authors integrating with the File Catalog are strongly encouraged to provide them.

Beyond these fields, individual applications may store additional information under their own key. For example the Java Archival and Data Exchange [7] (JADE) software stores additional metadata values under the key ‘jade’. The application author is not required to provide a schema, and may use it as a private metadata space. The expectation is that broadly useful values may eventually be promoted from these private spaces to well-defined optional top level fields as operational experience dictates.

2.4 Expansion of API

Another service created by IceCube is the Final Analysis Sample registration service. This service aims to support reproducibility of the scientific results in IceCube by uniquely identifying derived data products and the software used to produce them. The Final Analysis Sample service relies upon the File Catalog as a data store for the input and output files of published analyses. The initial REST API was expanded to include new routes to handle Collections and Snapshots.

2.4.1 Collections API

Collections are a set of files defined by a query. If a new metadata record added to the File Catalog matches the query criteria, it will appear in the Collection. The REST API routes added to handle Collections are as follows:

- GET /api/collections - Query the catalog for Collections
- POST /api/collections - Create a new Collection in the catalog

- GET /api/collections/{UUID} - Read a Collection from the catalog
- GET /api/collections/{UUID}/files - Read the files of a Collection

2.4.2 Snapshots API

Snapshots are a collection, captured at a specific point in time. If a new metadata record added to the File Catalog matches the query criteria of a Collection, it will be added to the Collection, but it will not appear in any Snapshots created before the new metadata record was added. Snapshots created after the new metadata record was added would include the new metadata record. The REST API routes added to handle Snapshots are as follows:

- GET /api/collections/{UUID}/snapshots - Query the catalog for Snapshots
- POST /api/collections/{UUID}/snapshots - Create a new Snapshot

- GET /api/snapshots/{UUID} - Read a Snapshot from the catalog
- GET /api/snapshots/{UUID}/files - Read the files of a Snapshot

3 Results

The initial REST API required writing about 800 lines of Python [8] code. The service was then made available to application developers, who used it to import metadata records from application-specific databases. 350 million records were imported into the File Catalog. The expansion of the REST API to include collections and snapshots required an additional 400 lines of Python code, bringing the File Catalog to a total of 1200 lines of Python.

3.1 Performance

By tuning the number of POST requests made in parallel to the REST API, the rate of document creation in the service reached a peak of 200 documents per second. This means it took about 3 weeks to import 350 million records. Although this is a significant amount of time, the initial import is a one-off event. It is not expected that this will need to be repeated in the future.

New records are not generated at rate even approaching 200 documents per second. This means the performance of the existing system, a single python service talking to a single MongoDB, is adequate to handle our metadata catalog needs. If there was pressure to increase performance, scaling up to a MongoDB cluster and adding more python services is relatively straightforward. This has not been explored because it is not necessary for our use-cases.

4 Discussion

The File Catalog is a successful service. The broadly supported technologies allowed easy integration with diverse applications written in Java, JavaScript, and Python. Shortly after deployment, the File Catalog became a dependency of the Final Analysis Sample registration service. It is anticipated that more services will integrate with or rely upon the File Catalog as they are developed.

4.1 Metadata schema

The downside to the flexible metadata schema is that each metadata record has a limited number of fields that can be relied upon. For example, the data transfer application may record which archival media contained a copy of a file. However this isn't a required field, so the data transfer application is not required to record this information, nor be consistent with how this information is recorded, in the File Catalog.

In the early phases of File Catalog implementation, the metadata schema was not well-defined beyond the required fields. The optional top-level fields were populated inconsistently, and it was recognized that lack of consistent data population is functionally equivalent to a private metadata space. In order to make the metadata broadly useful, across multiple applications, they must be populated in a consistent format and under consistent keys.

4.2 Future expansion

The File Catalog offers rich possibilities for expansion. Orthogonal routes for Collection, File, and Snapshot provide a convenient interface for new services. It is anticipated that a web service could be developed with relatively little effort, and allow a user-friendly interface to the File Catalog. The web interface could make popular queries, or queries around a well-known astronomical event, easy to run against the File Catalog on demand.

References

[1] The IceCube Neutrino Observatory: Instrumentation and Online Systems - IceCube Collaboration (Aartsen, M.G. et al.) JINST 12 (2017) no.03, P03012 arXiv:1612.05093 [astro-ph.IM]
[2] Ecma International. "ECMA-404—The JSON Data Interchange Format." (2013).
[3] Introducing JSON [Internet]. [cited 2018 Oct 3]. Available from: https://json.org/
[4] R. Fielding, and R. Taylor. Architectural styles and the design of network-based software architectures. **Vol 7**. Doctoral dissertation: University of California, Irvine (2000).
[5] MongoDB project, "MongoDB" [software], version 4.0.2, 2018. Available from: https://www.mongodb.com/download-center#community [accessed 2018-10-04]
[6] P. Leach, M. Mealling, and R. Salz. "RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace, 2005." [Internet]. [cited 2018 Oct 4]. Available from: http://www.ietf.org/rfc/rfc4122.txt
[7] P. Meade. "jade: An End-To-End Data Transfer and Catalog Tool." Journal of Physics: Conference Series. Vol. **898**. No. 6. IOP Publishing, 2017.
[8] Python project, "Python" [software], version 3.7.0, 2018. Available from https://www.python.org/downloads/release/python-370/ [accessed 2018-10-04]