Rucio Metadata SIG Update

Rob Barnsley
25/03/2024

Overview

Most of the work done so far for this SIG has concentrated on both extending the capabilities of Rucio's Plugin System and developing the Filtering Engine. Related metadata work that has been conducted in parallel is briefly discussed in Other. Prospective further work is discussed in Further Work.

Progress

Plugin System

Overview

The metadata plugin system existed before the formation of this SIG, but a significant amount of the work described here is so heavily reliant on the architecture of this system that it is worth describing briefly here.

The metadata plugin system is designed to be an easily extensible system for abstracting the handling (namely get/set/delete) of DID metadata by providing a stub abstract class (`rucio.core.did_meta_plugins.DidMetaPlugin` class) whose functions can be implemented to integrate new backend services.

To improve the plugin system's readability, the nomenclature of the plugin system was changed: references to the "HARDCODED" plugin have been superseded by the word "BASE", and references to "FALLBACK" have been superseded by the word "CUSTOM".

All metadata operations proceed through a sequence of plugins defined in the server's configuration file. This sequence always starts with the BASE plugin (which, as it is required by the application to function correctly, does not need to be explicitly configured): `rucio.core.did_meta_plugins.did_column_meta.DidColumnMeta`. This plugin allows for the setting of a restricted set of key/value pairs with specific types corresponding to (some of) the columns of the `dids` database table.

By default, Rucio assigns a CUSTOM metadata store that points to a JSON column in the application's own `did_meta` database table. This can also be explicitly defined by specifying the use of `rucio.core.did_meta_plugins.json_meta.JSONDidMeta` as one of the (comma separated) **plugins** in the **[metadata]** section of the server's configuration file. CUSTOM metadata plugins refer to any metadata plugins that can interface with Rucio's metadata plugin system allowing users to set custom key/value pairs.

The sequence of plugins specified in this configuration item is order-dependent; only one plugin can manage a key of a specific name. In the event that two plugins have keys with the same name, precedence for operations will be given to the first plugin in the sequence. The BASE plugin takes precedence over any CUSTOM plugins.

In the context of this SIG, work has recently been done on the creation of plugins for both MongoDB[1] and externally hosted PostgreSQL database instances.

Plugin for MongoDB (1.28 onwards, PR#5248)

A MongoDB instance can be connected by including `rucio.core.did_meta_plugins.mongo_meta.MongoDidMeta` in the list of **plugins** in the server configuration file and additionally specifying:

- the service host (`mongo_service_host`),
- the service port *(*`mongo_service_port`*)*,
- the database *(*`mongo_db`*)* and,
- the collection *(*`mongo_collection`*)* that the metadata should be stored in.

Plugin for external PostgreSQL (1.29 onwards, PR#5649)

An external PostgreSQL instance can be connected by including `rucio.core.did_meta_plugins.postgres_meta.ExternalPostgresJSONDidMeta` in the list of **plugins** in the server configuration file and additionally specifying:

- the postgres user (`postgres_user`),
- the postgres password (postgres_password),

[1] https://www.mongodb.com/

- the service host (`postgres_service_host`),
- the service port *(`postgres_service_port`)*,
- the database (`postgres_db`),
- the database schema (`postgres_db_schema`),
- the database table (`postgres_table`), and
- whether the table is to be managed by Rucio (`postgres_table_is_managed`).

Setting `postgres_table_is_managed` to “True” will request that Rucio manage the creation of the table. If an existing table with the correct schema exists, this should be set to “False”.

Filtering Engine (1.29 onwards, PR#4746)

As of 1.26, there was limited support for searching and retrieving DIDs based on metadata. Only AND’ed expressions using the equality operator were supported. The filtering engine was developed to enhance this functionality. It enables users to express complex conditions for DID filtering by constructing “filter” expressions that can be passed to `list dids` calls. These expressions must adhere to the following syntactic conventions:

- a semicolon (;) represents the logical OR operator,
- a comma (,) represents the logical AND operator, and
- all operators belong to set of (<=, >=, ==, !=, >, <, =).

Both one sided and compound inequalities are supported. Searching dates is also supported if the format is compliant with ISO 8601.

The filter engine translates these expressions into queries that can be executed against the corresponding plugin (all the keys referenced in the filtering expression must belong to the same plugin). Currently supported translations include SQLAlchemy, PostgreSQL and MongoDB.

Usage via the CLI

Filter expressions can be passed via the `--filter` flag of the `list-dids` command.

Usage via DIDClient

Programmatic interfacing with the filtering engine during calls to `list_dids` is also possible. The expected input to the `filter` parameter is a nested dictionary of key/value pairs like e.g. `[{'key1': 'value1', 'key2.lte': 'value2'}, {'key3.gte, 'value3'}]`. Keypairs in the same dictionary are AND'ed together, dictionaries are OR'ed together.

Keys to these pairs should be suffixed with `.<operation>`, e.g. `{'key1.gte': value}` is equivalent to `key1 >= value1`, where `<operation>` belongs to one of the set `{'lte', 'gte', 'gt', 'lt', 'ne' or ''}` (equivalence doesn't require an operator).

Other

Recent contributions[2] to the metadata component have also come from the BelleII[3] experiment. Specifically, these include:

- the option for metadata to be inherited from its parent container (PR#4819), and
- bulk getting and setting of metadata (PR#3677, PR#3678, PR#4014).

Additionally, BelleII have conducted some read/write benchmarking tests using the default JSON CUSTOM metadata plugin. Although the figures from these tests are dependent on the underlying infrastructure and setup/optimisation of the component services (particularly the database), they are promising and give some sense of what is possible. In short, write (setting) speeds of 1.160kHz and read (getting) speeds of 166Hz were achieved, resulting in a 70% CPU load on the Rucio frontend and ~3% load on the database backend. They note that this frontend load could be reduced by deploying multiple instances.

Further Work

Benchmarking

While the benchmarking tests conducted by BelleII offer valuable insights into the system's performance, advanced use cases often have more complex data retrieval and analysis requirements. Simple read and write operations may not adequately capture the demands of advanced data processing workflows, such as cross-referencing heterogeneous datasets or conducting sophisticated analytical queries.

Benchmarking these more complicated metadata query use cases would help in assessing the scalability and versatility of the metadata system, allowing communities to better discern if the system meets their requirements. Work in this area could include gathering use cases from a more diverse set of communities and implementing a framework to benchmark the queries that result from these use cases.

Optimisation

If the metadata benchmarking process identifies any issues within the system that are causing bottlenecks, the underlying indexing strategies and query execution plans could be evaluated and potentially improved to ensure that the system is performing optimally.

Plugins

---

[2] https://indico.cern.ch/event/1068644/contributions/4512194/attachments/2310792/3932294/2021-09-16%20-%20Metadata%20tests%20in%20Belle%20II.pdf

[3] https://www.belle2.org/

More plugins could be developed to support other backend technologies. For example, in-memory databases such as Redis[4].

[4] https://redis.io/