Design principles of the Metadata Querying Language (MQL) implemented in the ATLAS Metadata Interface (AMI) ecosystem

Jérôme Fulachier[1], *Jérôme* Odier[1], *Fabian* Lambert[1].

[1]Laboratoire de Physique Subatomique et de Cosmologie, Université Grenoble-Alpes, CNRS / IN2P3, 53 rue des Martyrs, 38026, Grenoble Cedex, FRANCE

Abstract. This document describes the design principles of the Metadata Querying Language (MQL) implemented in ATLAS Metadata Interface (AMI), a metadata-oriented domain-specific language allowing to query databases without knowing the relation between tables. With this simplified yet generic grammar, MQL permits writing complex queries more simply than with Structured Query Language (SQL).

1 Introduction

ATLAS Metadata Interface (AMI) is a generic ecosystem for metadata aggregation, transformation and cataloguing which benefits from about 20 years of feedback in the Large Hadron Collider (LHC) context. It is the official metadata repository for datasets and production parameters of the ATLAS [1] experiment. It implements Metadata Query Language (MQL), a metadata-oriented Domain-Specific Language (DSL) allowing to query databases without specifying all the relation between tables. With its simplified and generic grammar, MQL permits writing complex queries more simply than with SQL. This document describes how AMI compiles MQL into SQL queries using the underlying table relations graph automatically extracted through a reflection mechanism. A detailed description of the AMI framework design principles is available in earlier CHEP proceedings [2, 3, 4, 5, 6].

2 MQL Language

MQL is a domain specific language for executing queries on a Relational DataBase Management System (RDBMS) closely to spoken language. It is one of the main added-value features of AMI. Initially proposed by gLite [7], a middleware project for grid computing at LHC experiments, the specification was extended by the AMI team. MQL only deals with metadata entity names while SQL uses a catalog / table / field paradigm. It provides a way to produce SQL queries with a simplified syntax less prone to error. The MQL implementation in AMI is based on its database structure reflection subsystem. The language permits including or excluding path fragments in the relationship graph and is able to deal with table cycle dependencies.

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

2.1 Concepts and benefits

MQL queries are similar to SQL queries but easier to write. MQL queries are more user oriented than expert oriented. Even if the MQL syntax is simpler than the SQL one, it permits to execute the same kind of queries, with full control on the generated SQL. In particular, it will always be possible to write an MQL query reproducing the result of a given SQL query even if it is trading verbosity for more assumption about JOIN type.

For example, consider a physicist wants to list the names of the datasets that have files with size > 0.

In MQL, table relations are hidden:

```
    SELECT dataset.name WHERE file.size > 0
```

In SQL, it depends on the database schema (SQL joins):

```
    SELECT  dataset.name  FROM  dataset,  file  WHERE  dataset.id  =
file.datasetForeignKey AND file.size > 0
```

The MQL language does not contain any FROM clause nor join (on foreign keys). Nevertheless, even if joins are not necessary in MQL queries, the end-user can provide constraints on the relation paths. If no constraint is provided, the MQL to SQL algorithm follows all the possible paths of the relation graph. As a result, for complex database structures, it could generate slow or irrelevant SQL query. The way of specifying path constraints is described Section 2.3.

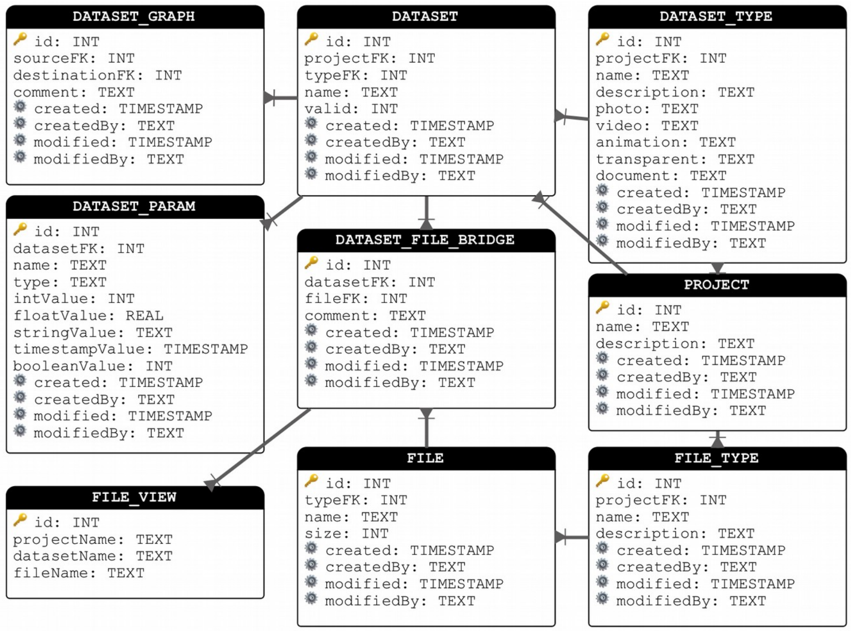

Figure 1. Sample database structure. In this schema, each table contains a unique auto-incremented identifier called "id". The "PROJECT" table contains projects information. This table doesn't have any foreign key (relation). The "DATASET" table contains entities representing a set of data. "DATASET" is linked to "PROJECT" with the foreign key constraint (`DATASET.PROJECTFK = PROJECT.ID`). The "DATASET_PARAM" table contains metadata parameters linked to the records of "DATASET". Those parameters have a name and a typed value. The "FILE" table is linked to "DATASET" thought the bridge to the "DATASET_FILE_BRIDGE" table. This table has two foreign key dependencies on both "DATASET" and "FILE". The "DATASET_TYPE" and "FILE_TYPE" tables introduce cycles in the relational dependency graph. Each of them have a foreign key dependency to the "PROJECT" table and to their related entity.

2.2 Relational model

In the following Figure 1, the relational model of a concrete use case is described, in order to expose various aspects of the MQL language and the resulting consequences on the generated SQL.

2.3 MQL specification

The MQL specification provides an interface to interact with any relational data source. It allows one to perform generic selection, insertion, modification and deletion operations, keeping benefits of the underlying relational model, but with a syntax less verbose of SQL.

MQL introduces the notion of basic Qualified Identifier (QId) for representing either an entity (aka a table), or a field. Its syntax is `[database.]entity` to a table and `[database.entity.]field` or `[entity.]field` for a field.

As previously explained, there is neither a FROM clause nor join expression in MQL. In most cases, it means that if there is no cycle in the relation graph, MQL is able to autogenerate both the FROM clause and join expressions. If there are cycles, it is necessary to indicate the paths to be included or excluded by specifying between braces, at QId level, a set of constraint QId (tables or fields). See Figure 2.

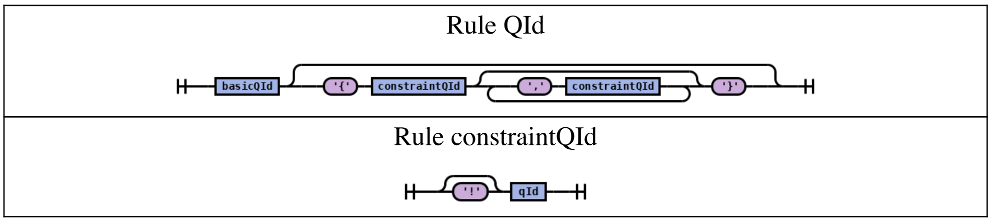

Figure 2. Grammar of a MQL Qualified Identifier (QId) for representing a table or a field. A "constraintQId" permits characterizing the path for resolving the table or the field. "!" is used to exclude a table or a field from the path.

MQL also introduces an isolation mechanism. It permits producing SQL embedded sub-queries in order to perform complex relational operations on the data source. This is why, in addition to normal expression groups, delimited with parentheses, MQL introduces isolated groups delimited with square brackets. See Figure 4. The effect is to isolate the condition from the main query and thus to restrict the paths used to reach the QIds contained in the condition to only this condition.

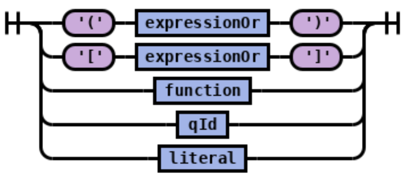

Figure 3. MQL has normal expression groups with parentheses and isolated expression groups with square brackets in order to produce SQL embedded sub-queries.

MQL has SQL-like SELECT, INSERT, UPDATE and DELETE statements. See Figure 4.

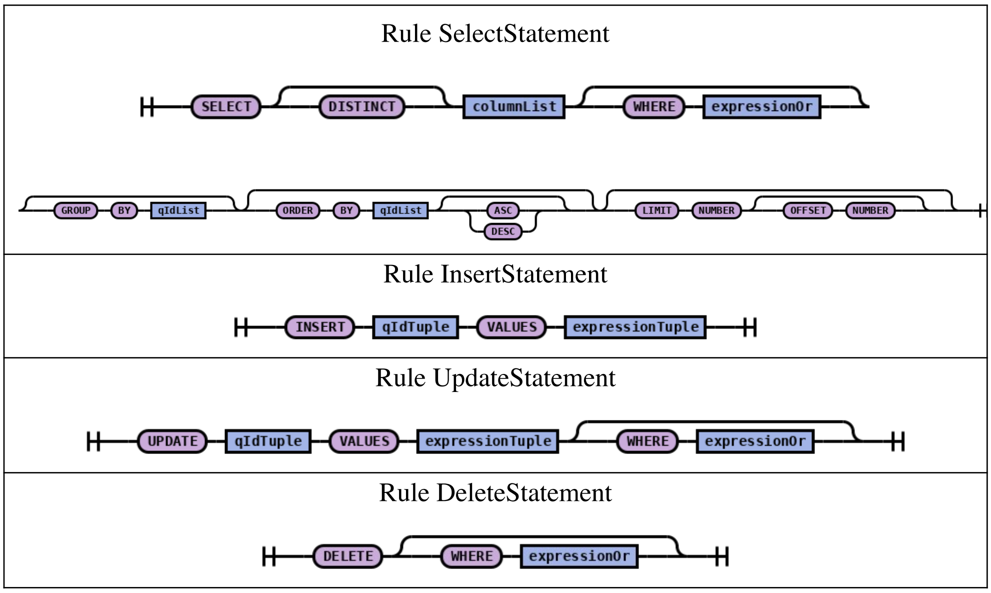

Figure 4. SQL-like SELECT, INSERT, UPDATE and DELETE statements implemented in MQL.

According to Figure 1, the query represented in Figure 5 searches for dataset names, having both “Xsection” > “x” and “Luminosity” < “y” and linked to non-empty files only through the “DATASET_FILE_BRIDGE” table (the paths reaching PROJECT.id are excluded by the {!PROJECT.id} constraint). Note that the two isolated conditions point to the same table.

```
SELECT DATASET.name
        WHERE
                [
                        DATASET_PARAM.name = 'Xsection'
                        AND
                        DATASET_PARAM.floatValue > x
                ]
        AND
                [
                        DATASET_PARAM.name = 'Luminosity'
                        AND
                        DATASET_PARAM.floatValue < y
                ]
        AND
                FILE.size{!PROJECT.id} > 0
```

Figure 5. Example of MQL query based on the database schema Section 2.2. If the constraint {!PROJECT.id} is not specified, the generated SQL will follow all possible paths to reach the “FILE” table.

3 SQL generation in AMI

After server-side processing, MQL queries are converted to optimized SQL queries before being executed on the data source by the appropriate JDBC [6] driver. This section describes the steps for generating SQL from MQL.

3.1 Reflection and paths resolution

The AMI framework has a JDBC based reflexion subsystem permitting the extraction of the database structure (tables, fields, foreign keys). It allows one to build a graph of relations which is put in a cache inside AMI. On top of this subsystem, AMI provides a mechanism to automatically resolve all the possible paths linking the QIds in a MQL query.

As an example, from Figure 1 the following relations can be resolved:
- Simple relation: DATASET <- DATASET_PARAM
- Bridge: DATASET <- DATASET_FILE_BRIDGE -> FILE
- Cycle: DATASET -> DATASET_TYPE -> PROJECT and DATASET -> PROJECT

3.2 MQL parsing and QIds resolution

The AMI MQL has its dedicated Java tokenizer and parser, autogenerated from a LL(*) grammar (top-down parsing) by using the Another Tool for Language Recognition (ANTLR) framework [8].

The parsing of an MQL query produces an Abstract Syntax Tree (AST) object containing all the necessary information to, later, generate a SQL query. During the parsing, all the QIds in the query are resolved (catalogs, tables, fields, relations) using the AMI reflexion sub-system taking into account the provided QId path constraints.

3.3 SQL Generation

Using the resolved query AST, AMI is able to build both the FROM clauses and SQL joins in the *WHERE* clauses. The isolated expressions are encapsulated into nested sud-SQL queries.

Figure 6 shows the generated SQL query for the MQL query. It is more verbose than the initial MQL, and this length ratio is even higher when the query becomes more complex.

```
SELECT DATASET.NAME
FROM     DATASET, FILE, DATASET_FILE_BRIDGE
WHERE
         DATASET.ID = DATASET_FILE_BRIDGE.DATASETFK
         AND
         FILE.ID = DATASET_FILE_BRIDGE.FILEFK
         AND
         DATASET.ID IN
                  (
                  SELECT DATASET.ID
                  FROM DATASET, DATSET_PARAM
```

```
                WHERE
                        DATASET.ID = DATASET_PARAM.DATASETFK
                        AND
                        DATASET_PARAM.NAME = 'Xsection'
                        AND
                        DATASET_PARAM.FLOATVALUE > x
                )
        AND
        DATASET.ID  IN
                (
                SELECT DATASET.ID
                FROM DATSET, DATASET_PARAM
                WHERE
                        DATASET.ID = DATASET_PARAM.PARAM.FILEFK
                        AND
                        DATASET_PARAM.NAME = 'Luminosity'
                        AND
                        DATASET_PARAM.FLOATVALUE < y
                )
        AND
                FILE.size > 0
```

Figure 6. SQL query generated form the MQL query of Figure 5.

Note that for performance reason AMI has an AST optimizer, transforming isolated expressions to non-isolated expressions if there is no ambiguity.

In a simple case: `SELECT * FROM A WHERE name='test'` is used instead of: `SELECT * FROM A WHERE A.id IN (SELECT id FROM A WHERE name='test').`

4 Conclusion

Both database experts and end-users can take advantage of the AMI MQL language. It provides the same features of SQL but with a lightweight syntax. For non-expert users, it can totally mask the database relations and gives the possibility to easily perform complex queries. The AMI Core Framework takes full benefit of MQL with almost no overhead compared to SQL, especially thanks to cache usage.
The new version of the AMI framework is in production in ATLAS and has already been chosen by other experiments for their metadata workflow.

5 Acknowledgments

Over the years, we have been helped and supported by many people at CC-IN2P3 and more recently by the Rosetta/Philae collaboration, in particular: Osman Aidel, Philippe Cheynet, Benoît Delaunay, Pierre-Etienne Macchi, Emil Obreshkov, Mattieu Puel, Yves Rogez, Mélodie Roudaud and Jean-René Rouet.

References

1. ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, *JINST* ***3*** *S08003* (2008), doi:10.1088/1748-0221/3/08/S08003

2. J. Odier, O. Aidel, S. Albrand, J. Fulachier, F. Lambert, *Evolution of the architecture of the ATLAS Metadata Interface (AMI)*, *J. Phys.: Conf. Ser.* ***664*** *042040* (2015), doi:10.1088/1742-6596/664/4/042040

3. J. Odier, O. Aidel, S. Albrand, J. Fulachier, F. Lambert, *Migration of the ATLAS Metadata Interface (AMI) to Web 2.0 and cloud*, *J. Phys.: Conf. Ser.* ***664*** *062044* (2015), doi:10.1088/1742-6596/664/6/062044

4. J. Fulachier, O. Aidel, S. Albrand, F. Lambert, *Looking back on 10 years of the ATLAS Metadata Interface*, *J. Phys.:* Conf. *Ser.* ***513*** *042019* (2013), doi:10.1088/1742-6596/513/4/042019

5. F. Lambert, J. Odier, J. Fulachier, *"Broadcasting dynamic metadata content to external web pages using AMI (ATLAS Metadata Interface) embeddable components"*, *EPJ Web Conf.* ***214*** *05046* (2019), doi:10.1051/epjconf/201921405046

6. "JDBC" [software]: https://www.oracle.com/database/technologies/appdev/jdbc.html [accessed 2020-01-10]

7. "gLite" [software]: http://grid-deployment.web.cern.ch/grid-deployment/glite-web/ [accessed 2020-01-23]

8. "ANTLR" [software]: http://www.antlr.org [accessed 2020-01-23]