MQT, an Approach for Runtime Query Translation: From EOL to SQL

Xabier De Carlos[1], Goiuria Sagardui[2], and Salvador Trujillo[1]

[1] IK4-Ikerlan Research Center, P. J.M. Arizmendiarrieta, 2 20500 Arrasate, Spain
`{xdecarlos,strujillo}@ikerlan.es`
[2] Mondragon Unibertsitatea, Goiru 2, 20500 Arrasate, Spain
`gsagardui@mondragon.edu`

Abstract. Managing models requires extracting information from them and modifying them, and this is performed through queries. Queries can be executed at the model or at the persistence-level. Both are complementary but while model-level queries are closer to modelling engineers, persistence-level queries are specific to the persistence technology and leverage its capabilities. This paper presents MQT, an approach that translates EOL (model-level queries) to SQL (persistence-level queries) at runtime. Runtime translation provides several benefits: (i) queries are executed only when the information is required; (ii) context and meta-model information is used to get more performant translated queries; and (iii) supports translating query programs using variables and dependant queries. Translation process used by MQT is described through two examples and we also evaluate performance of the approach.

Keywords: Model-Driven Development, Large-Scale Models, Runtime Query Translation, Model Queries

1 Introduction

Managing models encompasses tasks such as model validation, constraint checking, model analysis, etc. These tasks require executing queries over the models for getting information from them and also for modifying them. Model queries can be classified on two levels: model-level and persistence-level. Queries from both levels are complementary, but each level provides different benefits and limitations.

On the one hand, *model-level queries* are closer to modelling engineers since they are expressed in languages focused on interacting with models, independently of the persistence mechanism. Model-level query languages are for example Epsilon Object Language (EOL), Object Constraint Language (OCL), EMF Query, IncQuery, etc. Model-level query languages typically require the user to load the relevant models into memory first (e.g. using the `Resource` provided by EMF), before queries can be executed.

On the other hand, *persistence-level queries* are specific and dependent on a particular persistence mechanism. Some persistence-level query languages are for example: SQL for querying information persisted in relational databases; or MorsaQL [1] for querying models persisted using Morsa [2]. Persistence-level queries are typically executed directly over persisted models, without requiring to load model information first. Moreover, persistence-level languages leverage capabilities of persistence mechanisms for which they were created.

In this paper, we present the Model Query Translation layer (MQT), an approach that translates EOL queries (model-level) to SQL queries (persistence-level) at runtime. Runtime translation allows to translate imperative model-level queries that make use of variables and dependant queries. Moreover, runtime translation produces more performant queries through context and metamodel information.

The main contribution of this paper is MQT, a prototype that translates EOL queries to SQL. Using MQT, queries are translated and executed at runtime. We have evaluated it and the results show that the translation mechanism provided by MQT performs better (in terms of execution speed) than using the naive translation provided by Epsilon Model Connectivity Layer (EMC). Modification queries are not supported at this stage but we plan to add support for them in the future. Moreover, although currently MQT only translates EOL queries to SQL, in the future, we aim to extend it to support more model and persistence-level query languages.

The rest of the paper is organised as follows. In Section 2 we provide some background and motivation for this work. Section 3 presents MQT and illustrates the translation process through two examples. The approach is evaluated in Section 4, and Section 5 compares our approach with related work. We conclude the paper on Section 6 providing conclusions and directions for further work.

2 Background and Motivation

XMI is commonly used as a model persistence format in Eclipse Modelling Framework (EMF). Although other alternative file-based persistence solutions such as binary (supported by EMF) or JSON[3] exist, file-based persistence entails memory and performance problems with large models. When trying to solve scalability problems, most recent approaches [3,4,5,6] propose leveraging databases for large-scale model persistence. These approaches provide persistence-level query languages that leverage capabilities of these databases. This results in queries that are expressed at a low level of abstraction and tightly couples the queries with the specific model persistence mechanism used.

This scenario motivates us to provide a solution that automates query translation from persistence-agnostic model-level queries that are widely-used by modelling engineers to persistence-level queries that leverage the capabilities of the persistence layer to improve efficiency.

[3] Read more at `http://ghillairet.github.io/emfjson/`

3 MQT: Runtime Query Translation From EOL to SQL

Our work is focused on the implementation of a mechanism that allows querying models in a transparent way and using query languages closer to modelling engineers. Following these directions, we present MQT, an approach that provides automatic query translation from EOL, a model-level query language, to SQL, a persistence-level query language.

On the one hand, we have chosen EOL at model-level since it is an imperative OCL-like language that allows querying and modifying models of arbitrary modelling technologies. EOL provides interesting features such as: use of variables and methods, use of query chains and syntactic checking. Moreover, it is the base language of Epsilon Languages which provide different functionalities: model validation, model transformation, code generation, etc. On the other hand, we have chosen SQL at persistence-level since it is a structured and mature language widely used for performing queries in relational databases. However, it can be also used to query some NoSQL databases (e.g. using Unity JDBC[4] for running SQL queries against MongoDB databases).

We base our work on [7], where EOL is used to efficiently query large datasets stored on a single relational table. MQT supports translation of EOL query programs to SQL, and translated queries are only executed when the result is required. EOL is imperative and allows specifying variables and query chains. SQL does not support these features, which means that the expressive power of EOL and SQL is different. Consequently, the query translation cannot be total. Being so, in this case where the translated language (EOL) provides constructs that have no direct mapping in the target language (SQL), translation should be performed partially. The partial translation mechanism of MQT is based on the EMC, an API that provides abstraction facilities over modelling and data persistence technologies. To support the query translation, MQT prototype provides different classes that are described below:

- `MQTModel` It extends the `IModel` class provided by EMC and is the main class of the approach. Using this class MQT is able to interact with models conforming EMF in a uniform manner. In order to execute the SQL queries over databases where models are persisted, this class also implements a *jdbc* driver.
- `MQTResultObject` class. It implements the `IModelElement` interface of EMC and allows to work with each model element persisted within the database. This class translates, executes and get results of queries that ask about model elements and its features.
- `MQTResultSetList` class. This class translates, executes and get results of queries returning a list of `MQTResultObjects`.
- `MQTPrimitiveValueList` class. It implements the `IModelElement` interface and it is used to translate, execute and get results of SQL queries returning lists of primitive values (e.g. names).

[4] Read more at `http://www.unityjdbc.com/mongojdbc/mongo_jdbc.php`

Using these classes the approach is able to translate and execute at runtime EOL queries to SQL, and then get the results from a database where the queried model is persisted. However, the translation provided by EMC is **naive**: each query expression is translated and executed by the `MQTModel` and results of the SQL query are returned using `MQTResultObject`, `MQTResultSetList` or `MQTPrimitiveValueList`. Each query expression is translated and executed one-by-one. This occurs for example in the query ``‘‘EClass.all.select(...)’’``: first ``‘‘EClass.all’’`` is translated, returning a `MQTResultSetList` class with the result. Then, the ``‘‘.select(...)’’`` part of the query is translated, but using the `MQTResultSetList`. This requires to execute a SQL query for each result of the list to check the condition of the select.

But the previously described translation mechanism does not exhibit a good performance when translating and executing complex queries over large models, since it executes a lot of SQL queries. To improve scalability within query translation, MQT provides a mechanism that adapts translated SQL queries at runtime. To support the adaptive query translation `MQTResultSetList` class implements the `IAbstractOperationContributor` interface. In this way, naive translation of more complex queries such as selects, collects or rejects is replaced by our own translation mechanism. Using this mechanism MQT is able to group related and dependant EOL queries within a single translated SQL query. Being so, MQT executes fewer SQL queries (they are more accurate) and are executed only when the results are required.

We now explain the runtime query translation mechanism, by demonstrating the execution of the translation of different EOL programs. To avoid introducing a custom metamodel, sample query programs are executed over models that conform to the Ecore metamodel. As we have chosen SQL persistence-level query language, we have provided a metamodel-agnostic mechanism that persists models in a relational database. Relational databases require the specification of a schema where the structure of the database is defined. However, it is important to note that this paper is focused on the translation of queries and not on the efficient persistence of large-scale models. In order to facilitate understanding of the query translation process, the data-schema of the database is described previously.

3.1 Data-Schema

As the schema of Figure 1 illustrates, table *Object* persists elements of the model. This table contains *ObjectID* (primary key) identifying each element in the model and the *classID* (foreign key) that stores ID of its meta-class. Meta-classes are stored within the *Class* table, where *ClassID* (primary key) and *Name* are stored for each one. *Feature* table persists ids (*FeatureID*) and *Name*s of all the existing attributes and references. Values of attributes and references (*FeatureID*) of each model element (*ObjectID*) are stored in the *AttributeValue* and *ReferenceValue* tables (primitive value in case of attributes and id and meta-class in case of references). These five tables are enough for persisting the information stored in the models. However, sometimes adding duplicated information has a significant

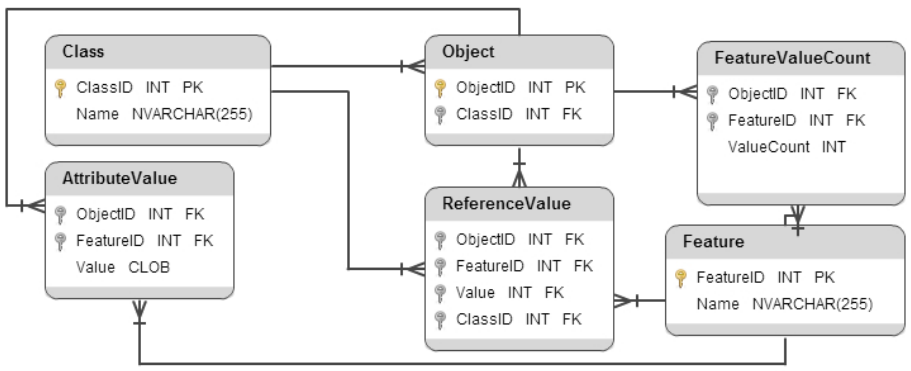

Fig. 1: Specified metamodel-agnostic database schema.

performance impact on the execution since the information can be used to get more effective translated queries (in terms of query execution speed). This is the case of *FeatureValueCount* table containing how many values each feature of each element has. Information of this table is used to get more effective translated queries: for example if *ValueCount* is 0, the feature do not exist for the element, and consequently, there is no need to query value of the feature; if *ValueCount* is X, it can be used to add `‘‘LIMIT X’’` expression within the translated SQL query.

To improve query execution speed, MQT loads some information of the database in the memory during query translation and execution. This is the case of `FeatureIDCache` and `ClassIDCache` (loading ID and name of the existing features and classes).

3.2 Translation Example 1: Simple queries

The first example, shown in Listing 1.1, illustrates a querying program which: retrieves all model elements that are instances of `EClass` and assigns only the first element of the list to the `class` variable (line 1); then, the program prints the value of the name attribute of the selected element (line 2); and finally, the program iterates elements that are referenced through the `eSuperTypes` reference of the selected element (line 3), printing the name of each one of them (line 4).

In the following paragraphs, we explain how MQT translates and executes these queries.

```
1  var class = EClass.all.first();
2  class.name.println();
3  for (superClass in class.eSuperTypes)
4    superClass.name.println();
```

Listing 1.1: Sample EOL program getting instances and attributes/references.

- **Line 1:** `EClass.all.first()`. This first query is divided in two parts. The first part is `EClass.all` and it creates a new instance of `MQTResultSet` adding the information required to query for all model elements that are instances of `EClass`. As such, behind the scenes, `MQTResultSet` executes the following SQL query to retrieve the list of matching elements from the database: `SELECT ObjectID FROM Object WHERE ClassID=?` The parameter of the query is `ClassID` obtained from memory using the previously introduced `ClassIDCache`. Then, the second part is executed (*first()* expression), returning only the first element (instance of `MQTResultObject`) of the `MQTResultSetList`, and it is stored in the *class* variable.
- **Line 2:** `class.name`. This query expression prints the name of the previously selected `MQTResultObject`. This is provided by a method implemented within the `MQTResultObject` (`getValue(feature)`) that retrieves attribute and reference values of the related model element from the database. In this case, the method executes the following query for getting the value of the name attribute: `SELECT Value FROM AttributeValue WHERE ObjectID = ? and FeatureID = ? LIMIT 1`. The query parameters are the `ObjectID` known by the `MQTResultObject` and the `FeatureID` obtained from memory (`FeatureIDCache`).
- **Line 3:** `for (superClass in class.eSuperTypes)`. This statement is similar to the previous but instead of returning an attribute value, a list of *ResultObjects* is returned and then iterated. The executed query is the following: `SELECT Value FROM ReferenceValue WHERE ObjectID = ? and FeatureID = ?`.
- **Line 4:** `superClass.name`. The execution logic of this statement is similar to the `class.name` statement.

3.3 Translation Example 2: Complex queries

Listing 1.2 shows another EOL program with more complex model element selection queries. In this example, the program first selects all the abstract EClasses without superclasses, and then computes the number of model elements that satisfy these conditions. If at least one element satisfies these conditions, the program prints the name of the first model element of the list.

```
var list = EClass.all.select(c|c.abstract=true)
var list2 = list.select(c|c.eSupertypes.isEmpty());
if(list2.size() > 0)
   list2.first().name.println();
```

Listing 1.2: Sample EOL program with selection.

Figure 2 illustrates the translation mechanism used for translating complex queries from Listing 1.2. Following, a more detailed description is provided:

- `EClass.all`. Creates a new instance of *MQTResultSetList* adding the information required to query all the model elements that are of the *EClass* meta-type.

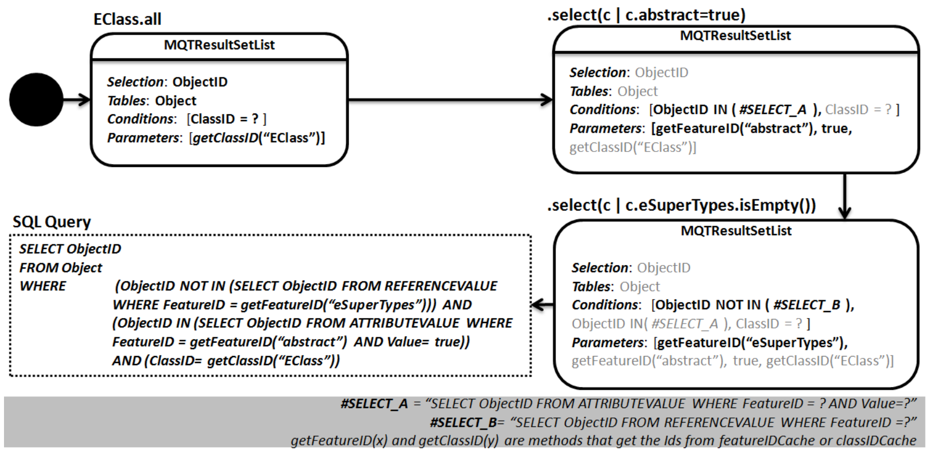
EClass.all
MQTResultSetList
Selection: ObjectID
Tables: Object
Conditions: [ClassID = ?]
Parameters: [getClassID("EClass")]
.select(c | c.abstract=true)
MQTResultSetList
Selection: ObjectID
Tables: Object
Conditions: [ObjectID IN (#SELECT_A), ClassID = ?]
Parameters: [getFeatureID("abstract"), true, getClassID("EClass")]
.select(c | c.eSuperTypes.isEmpty())
MQTResultSetList
Selection: ObjectID
Tables: Object
Conditions: [ObjectID NOT IN (#SELECT_B), ObjectID IN(#SELECT_A), ClassID = ?]
Parameters: [getFeatureID("eSuperTypes"), getFeatureID("abstract"), true, getClassID("EClass")]
SQL Query
SELECT ObjectID
FROM Object
WHERE (ObjectID NOT IN (SELECT ObjectID FROM REFERENCEVALUE WHERE FeatureID = getFeatureID("eSuperTypes"))) AND (ObjectID IN (SELECT ObjectID FROM ATTRIBUTEVALUE WHERE FeatureID = getFeatureID("abstract") AND Value= true)) AND (ClassID= getClassID("EClass"))
#SELECT_A = "SELECT ObjectID FROM ATTRIBUTEVALUE WHERE FeatureID = ? AND Value=?"
#SELECT_B= "SELECT ObjectID FROM REFERENCEVALUE WHERE FeatureID =?"
getFeatureID(x) and getClassID(y) are methods that get the Ids from featureIDCache or classIDCache

Fig. 2: Example of the runtime translation of the query from EOL to SQL.

- `select(c|c.abstract=true)`. This second expression completes the previous one. Being so, the same instance of `MQTResultSetList` is completed by adding a new condition and two parameters to get only the model elements that have the abstract feature with the true value.
- `select(c|c.eSuperTypes.isEmpty())`. This select expression completes more the `MQTResultSetList` instance. It adds a new condition and parameter to get only the model elements that do not have `eSuperTypes`.
- Finally, the information that has been added during the previous steps is used by the `MQTResultSetList` to get the translated SQL query.

MQT only executes the translated SQL query over the database when the results are needed. In the case of the previous example (Listing 1.2), the instance of `MQTResultSetList` executes the translated SQL in the line 3, where the size value needs be returned. Once the query described on Figure 2 is executed, `MQTResultSetList` instance obtains the results. Then `MQTResultSetList` provides size through a method that counts the quantity of the returned elements. In the case of line 4, `MQTResultSetList` uses same results but it returns only the first element of the list.

4 Evaluation

To evaluate MQT, we have executed an EOL query program over five models of different sizes (from 45MB to 403MB). All experiments have been executed using an Intel Core i7-3520M CPU at 2.90 GHz with 8GB of physical RAM running Windows 7 SP1 64bit, JVM 1.7.0 and the Eclipse Kepler SR1 distribution configured with 2GB of maximum heap size. Models have been created using MoDisco's Java Discoverer [8] and they specify source code of different Java plug-ins. These models have been persisted in a relational database with the previously described metamodel-agnostic schema and using the H2[5] DBMS (version 1.3.168). We have specified an EOL Query Program based on the GraBats'09 reverse engineering contest, where the query identifies singleton classes within source code of Java plug-ins specified by the models. We have used GraBats'09 because is widely used by the community on the studies related to model persistence.

EOL Query program has been executed 100 times over each model and using the naive translation of EMC and the MQT translation (previously explained on Section 3). Table 1 shows information of the models used during experimentations: *size*, number of *objects*, number of *methods* and number of *singleton classes* that they contain.

Table 1: Singleton query program execution over different models.

	Size	Objects	Methods	Singl. Classes	Naive trans.	MQT trans.
M1	45MB	165741	5366	9	185ms	13ms
M2	72MB	330761	8129	8	302ms	11ms
M3	212MB	875988	11393	6	676ms	10ms
M4	327MB	1343207	15386	0	950ms	3ms
M5	403MB	1566890	19366	0	1243ms	1ms

Average of the query translation and execution times obtained during experimentations are illustrated on *Naive translation* and *MQT translation* columns. As is shown on the table, with naive translation, model size has great impact on the required time for executing the query program and it increases as the size of the model increases. However, using MQT translation model size has less impact over the execution time. We can conclude with these results that MQT provides more scalability if queries are executed over large-models.

To assess the performance of MQT, we executed the same query against XMI models (provided by EOL) and obtained the same results. We have also analysed the execution time spent on the query translation: (M1) 3.35ms; (M2) 4,51ms; (M3) 0.77ms; (M4) 0.8ms; and (M5) 0.64ms. From these results, we have concluded that as the translation time is only few milliseconds it does not imply a temporary overload.

5 Related Work

Several solutions have been proposed in terms of generation of queries based on OCL-like languages. [9] proposes an approach focused on generating SQL queries from invariants specified using OCL. The approach allows mapping Unified Modelling Language (UML) models to other data schemas like databases

[5] More information about H2 at `http://www.h2database.com/`

and then generating queries that allow to evaluate invariants using SQL. [10] describes an approach that generates views using OCL constraints, and then uses these view to check the integrity of the persisted data. The approach has been implemented in OCL2SQL[6], a tool that generates SQL queries from OCL constraints. A similar approach for integrity checking is proposed in [11]. While these approaches are focused on translating OCL constraints into SQL queries at compile-time, our approach generates SQL queries from OCL-like expressions (EOL) at runtime. Comparing with compilation-time translation, main benefits of runtime translation are: (i) translated queries are executed only when the information is required; (ii) context and metamodel information can be used during query translation; and (iii) it supports query chains and variables within the query program.

[12] presents SPARQLAS, an SPARQL-like query syntax that is translated to SPARQL and then executed against OWL knowledge base, using results as input for OCL queries. Using SPARQLAS, queries are executed using SPARQL (persistence-level) and then query results are the input of OCL queries (model-level). By contrast, our approach translates queries from EOL (model-level) to SQL (persistence-level) and then executes the obtained SQL queries.

6 Conclusions and Further Work

In this paper we have presented MQT, an approach that translates at runtime and automatically model queries specified using EOL (model-level query language) to SQL (persistence-level query language). Main benefits of the runtime translation are: (i) on-demand execution of translated queries; (ii) queries are translated and adapted at runtime and metamodel and context is used to get more effective translated queries; (iii) it allows to execute query programs with variables and dependant queries.

We have evaluated our approach, concluding that with the runtime translation and adaptation of queries, MQT provides a scalable solution to query large-models. However, for the future, we plan to perform a more complete evaluation that analyses the impact of the characteristics of the used models and queries.

Presented prototype supports all the EOL expressions that obtain information from models, but modification expressions are not supported. For a next version, we plan to extend the approach with support for: (i) modification queries; (ii) additional model-level query languages (e.g. OCL); and (iii) additional persistence-level query languages (e.g. Cypher). This will provide a solution that allows engineers to write high-performance queries in a model-level query language without worrying about the model persistence format. Regarding this point, an open issue to be analysed is how to provide extensibility to facilitate the integration of query languages at both sides.

[6] Read more at `http://dresden-ocl.sourceforge.net/usage/ocl22sql/`

Acknowledgments

Authors wish to thank Dr. Dimitris Kolovos for his help on this work. This work is partially supported by the EC, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (#611125).

References

1. Pagán, J.E., Molina, J.G.: Querying Large Models Efficiently. Information and Software Technology **56**(6) (2014) 586 – 622
2. Espinazo Pagán, J., Sánchez Cuadrado, J., García Molina, J.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. In Whittle, J., Clark, T., Khne, T., eds.: Model Driven Engineering Languages and Systems. Volume 6981 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 77–92
3. Eike Stepper: CDO Model Repository Overview. `http://www.eclipse.org/cdo/documentation/` Accessed March 17, 2014.
4. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A Repository for Scalable Model Management. Software & Systems Modeling (2013) 1–21
5. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a Scalable Persistence Layer for EMF Models. In: ECMFA- European conference on Modeling Foundations and applications, York, UK, Royaume-Uni, Springer (July 2014)
6. Scheidgen, M.: Reference Representation Techniques for Large Models. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. BigMDE '13, New York, NY, USA, ACM (2013) 5:1–5:9
7. Kolovos, D.S., Wei, R., Barmpis, K.: An Approach for Efficient Querying of Large Relational Datasets with OCL-based Languages. In: XM 2013–Extreme Modeling Workshop. (2013) 48
8. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10, New York, NY, USA, ACM (2010) 173–174
9. Heidenreich, F., Wende, C., Demuth, B.: A Framework for Generating Query Language Code from OCL Invariants. ECEASST **9** (2008)
10. Demuth, B., Hussmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In Gogolla, M., Kobryn, C., eds.: UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Volume 2185 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 104–117
11. Marder, U., Ritter, N., Steiert, H.: A DBMS-Based Approach for Automatic Checking of OCL Constraints. In: Proceedings of Rigourous Modeling and Analysis with the UML: Challenges and Limitations. OOPSLA (1999)
12. Parreiras, F.S.: Semantic Web and Model-driven Engineering. John Wiley & Sons (2012)