Model Query Translator
A Model-level Query Approach for Large-scale Models

Xabier De Carlos[1], Goiuria Sagardui[2], Aitor Murguzur[1], Salvador Trujillo[1] and Xabier Mendialdua[1]
[1]*IK4-Ikerlan Research Center, P .J. M. Arizmendiarrieta, 2 20500 Arrasate, Spain*
[2]*Mondragon Unibertsitatea, Goiru 2, 20500 Arrasate, Spain*

Keywords: Model-Driven Development, Large-scale Models, Query Languages, Persistence, Eclipse Modelling Framework, Scalable-query.

Abstract: Persisting and querying models larger than a few tens of megabytes using XMI introduces a significant time and memory footprint overhead to MDD workflows. In this paper, we present an approach that attempts to address this issue using an embedded relational database as an alternative persistence layer for EMF models, and runtime translation of OCL-like expressions for efficiently querying such models. We have performed an empirical study of the approach using a set of large-scale reverse engineered models and queries from the Grabats 2009 Reverse Engineering Contest. Main contribution of this paper is the Model Query Translator, an approach that translates (and executes) at runtime queries from model-level (EOL) to persistence-level (SQL).

1 INTRODUCTION

Model Driven Development (MDD) raises the level of abstraction from code to models and makes the latter first-class citizens of the development process. In some domains, models used for MDD can become very large. For example, in embedded system domains such as wind-power or railway, systems can comprise a large number of elements such as sensors, actuators and control units. To effectively support such domains, scalable model persistence mechanisms are essential. Unfortunately, this is not the case for the standard XML Metadata Interchange (XMI), which the Eclipse Modelling Framework (EMF) uses as its default persistence format (Pagán et al., 2013).

To overcome scalability problems several approaches have been proposed to leverage relational and non-relational databases to facilitate scalable model persistence and loading. Those provide an alternative persistence mechanism for large-scale models entailing different elements: (i) providing a persistence approach with features that facilitate scalability such as partial loading or loading on demand; (ii) integrating the persistence mechanism within commonly used modelling tools; and (iii) providing scalable querying of persisted models.

This paper focuses on the scalable querying of persisted models. For that purpose, we present the Model Query Translator (MQT) approach in order to translate at runtime queries expressed in the Object Constraint Language (OCL)-based Epsilon Object Language (EOL) to SQL. MQT supports read-only EOL queries and we plan to add support for modification query expressions in a next version. General overview of MQT was presented in (De Carlos et al., 2014). In this paper we provide technical details of the solution and an empirical study to compare it with XMI.

With MQT users can execute queries using the same level of abstraction used for querying models persisted using XMI, but with the efficiency of SQL. As a quantitative evaluation, we have performed an empirical study using five models of different sizes (from 45.3MB to 403MB). Each model has been persisted using both XMI and our persistence approach (embedded database). Then we have executed the *GraBaTs'09 Reverse Engineering Contest* query (singleton extraction). In our experiments, models persisted using our approach take up more storage space but consistently outperform their XMI counterparts in terms of memory footprint and query execution time.

62 De Carlos X., Sagardui G., Murguzur A., Trujillo S. and Mendialdua X..
Model Query Translator - A Model-level Query Approach for Large-scale Models.
DOI: 10.5220/0005238000620073
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 62-73
ISBN: 978-989-758-083-3
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

Roadmap of the Paper

The rest of the paper is organised as follows: Section 2 provides some background and motivates this work. Sections 3 and 4 describe MQT approach and the translation process. An empirical study is performed in Section 5 and then Section 6 reviews related work and compares it with the proposed approach. This paper concludes with conclusions and directions for future work are proposed in Section 7.

2 BACKGROUND AND MOTIVATION

XMI is an XML-based model interchange format standardised by the OMG, and the default model persistence format in the widely-used EMF. In addition to XMI other file-based persistence formats exist: binary (supported by EMF) or JSON[1]. However, file-based persistence entails memory and performance problems with large models, since the information needs to be fully loaded in memory before it can be queried (Pagán et al., 2013). As models grow in size, this can have a significant impact both on the overall time needed to execute a query on a stored model, and on the memory footprint of the host application.

Most recent approaches (Eike Stepper, 2014; Pagán et al., 2013; Benelallam et al., 2014; Scheidgen, 2013) try to solve scalability problems through persisting models in databases. These approaches provide persistence-level query languages that leverage the capabilities of these databases (e.g. SQL, MorsaQL (Pagán and Molina, 2014), etc.). Persistence-level queries are specific and dependent on a particular persistence mechanism. This results in queries that are expressed at a low level of abstraction and tightly couples the queries with the specific model persistence mechanism. One of the advantages of persistence-level queries is that they are typically executed directly over persisted models. By contrast, model-level queries are closer to model engineers since they are expressed in languages focused on interacting with models, independently of the persistence mechanism (e.g. IncQuery, EOL, etc.). The main disadvantage of model-level query languages is that they typically require to load models into memory before queries can be executed.

This scenario motivates us to provide a solution that is able to query persisted models using a model-level query language, but also leveraging the capabilities of the persistence mechanism used.

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

3 MQT: OVERVIEW

MQT is an approach that supports querying models with a model-level language but also takes advantage of the persistence-level query language. MQT translates at runtime model-level (EOL) queries to persistence-level (SQL) queries.

We have chosen EOL as model-level query language. Main reasons for choosing EOL are that (i) EOL is a OCL-like language that besides read-only queries it also provides queries for model modification; (ii) other languages such as Epsilon Validation Language (EVL), Epsilon Transformation Language (ETL), Epsilon Comparison Language (ECL) are built on top of EOL (Kolovos et al., 2014) and provide model validation, transformation or comparison features. On the persistence-level, we have chosen SQL since it is a structured and mature language used to query information from relational databases (and also some NoSQL databases). EOL is imperative and cannot be directly mapped to SQL. For this reason, MQT performs partial translation of EOL queries into equivalent SQL queries. Partial translation is suitable when the model-level query language provides constructs that have no direct mapping in the target language. Approaches such as (Demuth et al., 2001) and (Marder et al., 1999) perform OCL to SQL translation at compilation-time, but the translation on MQT is executed at runtime. Runtime-translation facilitates translation of languages that have not direct mapping (e.g. EOL and SQL).

The translation mechanism provided by our approach is based on the Epsilon Model Connectivity Layer (EMC) (Kolovos et al., 2014). EMC is an API that provides abstraction facilities over modelling and data persistence technologies. It defines the `IModel` interface that provides methods that enable querying and modifying model elements. MQT provides, `EDBObject` class (described in Section 3.2) that implements the `IModel` interface of EMC. Using instances of this class, MQT is able to interact with models conforming to an Ecore metamodel and persisted in a relational database. Our approach is based on (Kolovos et al., 2013), where the naive translation provided by EMC is used to query large datasets stored on a single-table relational database. By contrast, MQT provides: (i) a metamodel-agnostic dataschema that is able to persist models conforming to any Ecore metamodel; and (ii) customized translation of queries from EOL to SQL. At this stage, although translation of read-only EOL query expressions is supported, MQT does not support query expressions that modify model.

3.1 Data-schema

We have chosen SQL query language on persistence-level. Consequently, translated queries must be based on a data-schema. We have specified a data-schema that is able to persist models in a relational database. The schema is metamodel-agnostic and persistence of models in the database is independent of meta-models they conform to. Thus, any model can be persisted under the same schema, so in case meta-model evolves, no changes are required in the schema. We have defined different indexes within the schema that allow running the translated SQL queries faster. Figure 1 illustrates the schema. The EOL to SQL translation process of MQT is dependent on this data-schema. Tables, relations and indexes shown on the schema are discussed below:

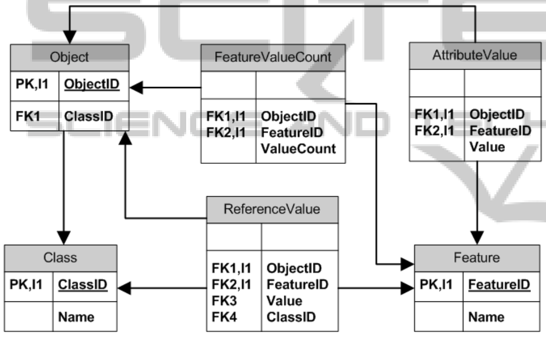


Figure 1: Specified metamodel-agnostic database schema.

- **`Object`** table: A tuple in this table is created for each element of the model. Model elements are identified by a primary key in the row of the `ObjectID` column and the meta-class ID (foreign key) of each element is stored in the row of the `ClassID` column. `ObjectID` has been defined as an index of this table.
- **`Class`** table: It contains all meta-classes of the model. `ClassID` (primary key) and `Name` of the meta-class are stored for each one. `ClassID` has been defined as an index of this table.
- **`Feature`** table: It stores an ID (`FeatureID` column, primary key) and the name (`Name` column) for each attribute and reference in the meta-classes of the model. `FeatureID` has been defined as an index of this table.
- **`AttributeValue`** table: This table stores attribute values of model elements. Attribute values are identified by an `ObjectID` and a `FeatureID` (both foreign keys), and the `Value` column stores the primitive value of the attribute. In case of single-file attributes with empty value, value-by-default is stored. An index has been created for this table on `ObjectID` and `FeatureID`.
- **`ReferenceValue`** table: This table stores references of model elements. References are identified by an `ObjectID` and a `FeatureID` (both foreign keys). The ID of the referenced element (Value column, foreign key) and meta-class of the referenced element (`ClassID` column, foreign key) identify the referenced model element. An index has been created for this table on `ObjectID` and `FeatureID`.
- **`FeatureValueCount`** table: Stores the number of values of each feature. Tuples are identified by two foreign keys (`ObjectID` and `FeatureID`), and the count is stored in `ValueCount`. This table is not indispensable to save the model information, but preliminary tests show that using it performance is improved in terms of speed. An index has been created for this table on `ObjectID` and `FeatureID`.

Although the schema is metamodel-agnostic, the metamodel is provided by the users when the EDB-Model class is instantiated (through a parameter on the constructor). Being so, MQT makes use of the metamodel to: (i) identify superclasses and subclasses of model elements; (ii) obtain the default value of each attribute; (iii) identify whether a feature is an attribute or a reference; or (iv) obtain bounds of features.

Additionally, we have provided an utility that giving a model persisted in XMI, returns a database with the previously described schema and containing all the model information (`es.ikerlan.edbm.EDBModelImport.importFromXMI(...)`).

3.2 Conceptual Model

Figure 2 illustrates a UML class diagram of MQT, where the conceptual model of MQT is specified. Next, each class is described one-by-one:

- **`EDBConnection.`** This class is responsible for connecting with the database and it is used to execute queries and get results from the database. The instance of `EDBConnection` contains the information required to connect to the database: username, password, basepath, database name and an instance of the JDBC Driver.
- **`EDBCache.`** Each instance of this class contains two `Maps` that keep in memory names and ids of classes and features in the queried model. `Maps` are accessible using different methods: `getClassID(name)`, `getFeatureID(name)`,

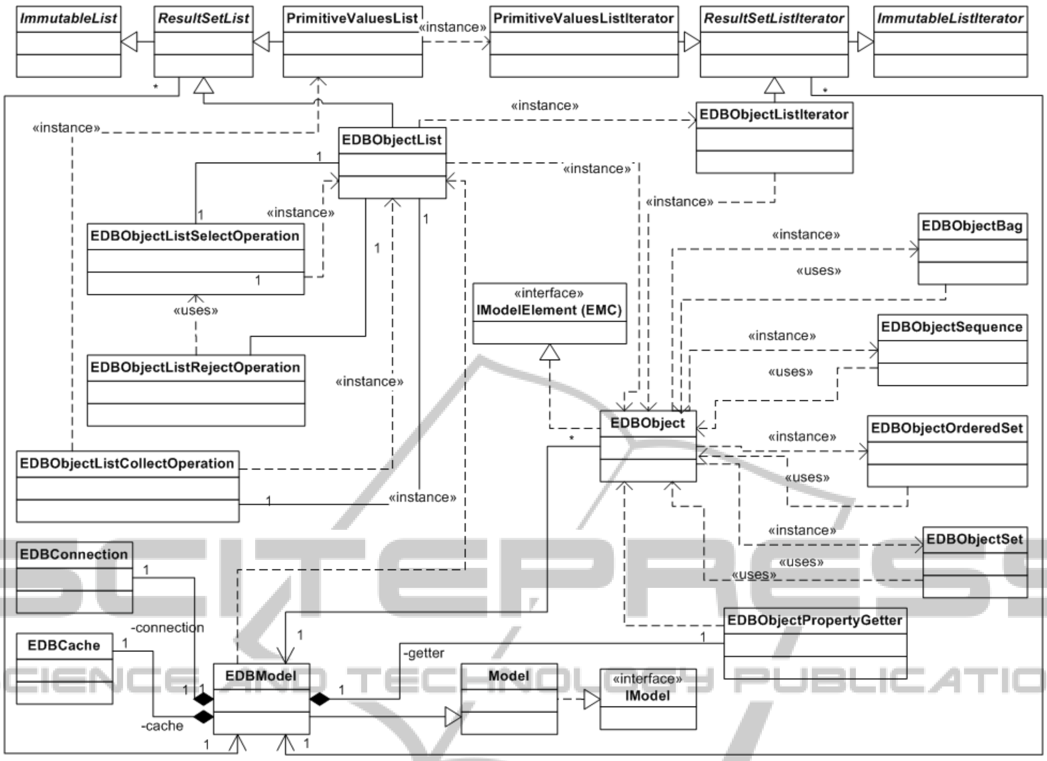


Figure 2: Class-diagram of MQT.

getClassName(id) or getFeatureName(id). Using methods provided by this class, it is possible to avoid some *joins* on the translated SQL queries, since names or ids of classes and features are obtained directly from memory.

- **EDBModel.** This class extends the Model class of EOL (which implements the IModel interface of EMC). This is the main class of MQT and instances of this class are responsible for interacting with model elements persisted in the database. Each instance of this class stores and uses an instance of the class EDBConnection and another one of EDBCache. The EDBModel class overrides Model methods that query models (modification methods are not supported). This class also overrides the getPropertyGetter() methods of Model, returning an instance of EDBObjectPropertyGetter.
- **EDBObjectPropertyGetter.** This class extends the AbstractPropertyGetter class of EOL. It is invoked when a feature value is queried, and it executes the method getValue(feature) of the queried EDBObject instance (the method is described in the next point).
- **EDBObject.** This class implements the IModelElement interface of EOL and it is used to represent model elements that are persisted in the database. EDBObject class is instantiated by classes such as EDBObjectList, EDBObjectIterator or EDBModel to return elements of the model. Instances of EDBObject class are able to get attribute and reference values of the model object that they represent. Attribute and reference values are obtained through the method getValue(String featureName). It also provides other methods to obtain other information about the model element: equals(Object object) compares the model element with another object and returns if it is equal; getEClass() returns the EClass of the represented model element; getOwningModel() returns the model where the element is contained; getObjectID() returns the ID that identifies the model element within the database; etc.
- **EDBObjectBag, EDBObjectSequence, EDBObjectSet, EDBObjectOrderedSet.** These classes extend different collection type classes of EOL (EolBag, EolSequence, EolSet and EolOrderedSet). These classes are instantiated and returned in the execution of the method getValue(feature) of the class EDBObject.
- **ImmutableList.** An abstract class that implements the interface java.util.List and pro-

vides support for read-only lists.

- **`ResultSetList`.** An abstract class that extends the previously described `ImmutableList`. This class is used to provide a `List` that is able to work with results returned by the database (`ResultSet`) after executing a SQL query.
- **`EDBObjectList`.** This class extends `ResultSetList` and specifies a list composed by `EDBObjects`. This class is instantiated when the translated and executed SQL query returns a list of model elements. List methods implemented in this class use a database (`ResultSet`) to return results. For example: the `size()` method returns the size of the `ResultSet`; `contains(Object object)` returns if the given object exists on the results of the `ResultSet`; etc. To support runtime adaptation of queries the `EDBObjectList` class implements the interface `IAbstractOperationContributor` and an implementation of `getAbstractOperation(String operation)` method is provided. Depending on the translated operation (select, collect, etc.) it returns an instance of class `EDBObjectSelectOperation`, `EDBObjectCollectOperation` or `EDBObjectRejectOperation`.
- **`PrimitiveValuesList`.** This class extends `ResultSetList` and is similar to the previously described `EDBObjectList`. In this case, this class is instantiated when the translated and executed SQL query returns a list of primitive values (instead of model elements).
- **`ImmutableListIterator`.** An abstract class that implements the interface `java.util.ListIterator` and provides an implementation for read-only list iterators.
- **`ResultSetListIterator`.** Abstract class extending the previously described `ImmutableListIterator`. This class is used to provide a `ListIterator` that is able to work with results returned by the database (`ResultSet`) after executing a SQL query.
- **`EDBObjectListIterator`.** Extends the class `ResultSetListIterator` and implements `IteratorList` methods but returning results (`EDBObjects`) based on a database (`ResultSet`). This class is instantiated when the method `listIterator()` of the `EDBObjectList` class is executed.
- **`PrimitiveValuesListIterator`.** Implements the same methods of the previous class but in this case it is adapted to work with a `ResultSet` containing primitive values. This class is instantiated when the method `listIterator()` of the `PrimitiveValuesList` class is executed.
- **`EDBObjectSelectOperation`, `EDBObjectCollectOperation` and `EDBObjectRejectOperation`.** These classes extend different `EOLOperations`, overriding the `execute()` method and providing more performant and adapted translated select, collect and reject queries. EMC provides a naive translation where each expression is translated and then executed one-by-one. For example, the expression ``` ``Elem.all.collect(n | n.name)'' ``` returns a list containing names of all model elements of type `Elem`. In the case of the EMCs' naive translation and execution of the query, first ``` ``Elem.all'' ``` is executed, getting a `EDBObjectList` containing all `Elem` type elements. Then, one SQL query is generated and executed to get the name of each element returned on the list. By contrast, extending `EOLOperations`, more performant queries are constructed when translating select, reject and collect expressions. For example, in case of ``` ``Elem.all.collect(n | n.name)'' ```, we can construct a SQL query that returns in one execution all Elem type elements' name. We compared query execution times of the naive and of the extended translation in (De Carlos et al., 2014).

4 MQT: EOL to SQL Translation

This section describes the translation process by analysing how MQT translates EOL query expressions. The EOL query parsing and execution is driven by the *EOL Module* and our approach is able to interact with it since MQT is based on the EMC. Figure 3 illustrates the sequence diagram of the translation of the most noteworthy EOL expressions. Following, the translation and execution process of the expressions illustrated in the Figure 3 and of other EOL expressions is described. We have extracted from (Kolovos et al., 2014) the description of each EOL expression .

4.1 Query Model Elements

The model itself is the input of EOL expressions of this type. Being so, the starting point of the translation are the methods located on the `EDBModel` class instance and they are executed by the *EOL Module*. First step of the Figure 3 illustrates the translation process of an EOL query of this type. Next, the transla-

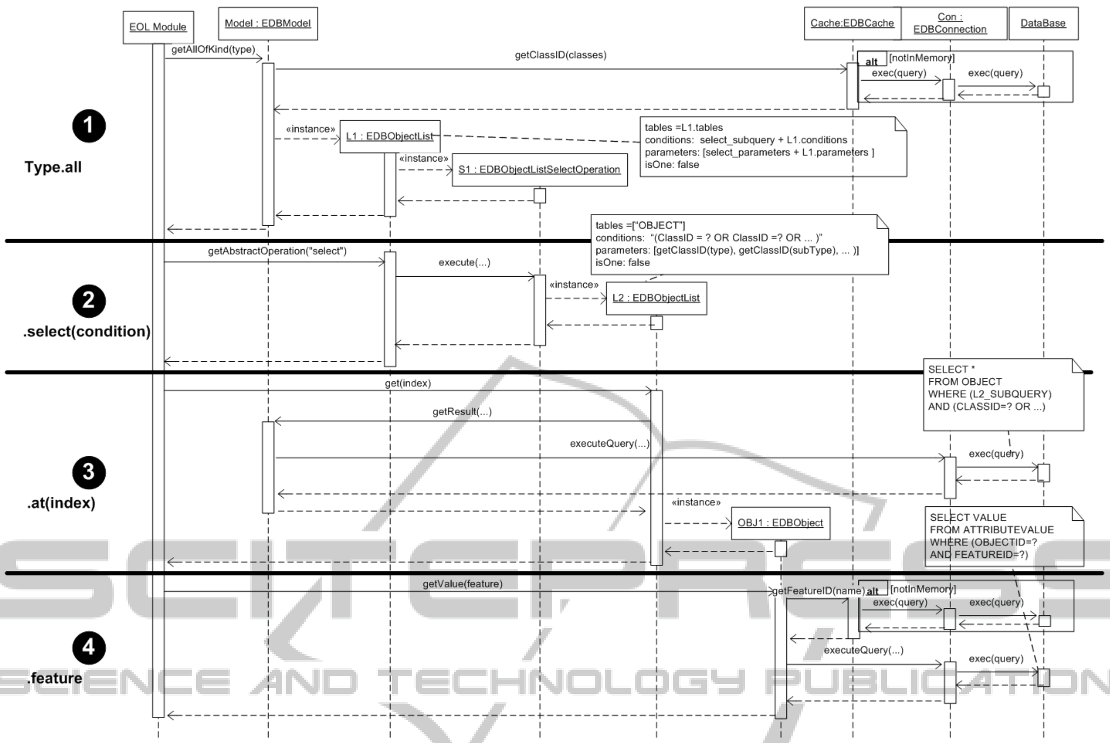


Figure 3: Sequence diagram of a MQT execution.

tion process for EOL expressions of this type is described:

- **`allOfKind(), allInstances(), all():EDBObjectList`**:

  EOL Description: `allOfKind()` returns an `EDBObjectList` containing all the model elements (`EDBObject` instances) that are instances either of the type itself or of one of its subtypes. Methods `allInstances()` and `all()` are aliases of `allOfKind()` and the execution and result is the same.

  MQT Translation: returned `EDBObjectList` is able to query from the database model elements that are instances of the type or of the subtypes: (`SELECT * FROM OBJECT WHERE (OBJECT.CLASSID = getClassID(type) OR OBJECT.CLASSID = getClassID(subtype) OR ...)`), but the query is not executed until the result is required. As step 1 of Figure 3 illustrates, when an `allOfKind()` EOL query is executed, the `EOLModule` executes the `getAllOfKind(String type)` method of the `EDBModel` class instance (*Model* in the figure). In this method, summarizing, first `classIds` of the type and subtypes (if exist) are obtained from memory through the `EDBCache` class instance. If the queried class is loaded in memory, it returns the ID directly. If it is not yet loaded, the ID of the class is obtained through a SQL query that is executed by an `EDBConnection` instance. Next, an `EDBObjectList` is instantiated (*L1*), where some of the constructor parameters are : `tables = ``[Object]'', conditions=``(ClassID=? OR ClassID=? OR ...)'' parameters=[getClassID(type), getClassID(subtype),...], isOne=false`. These parameters are used to construct the SQL query.

- **`allOfType():EDBObjectList`**:

  EOL Description: returns an `EDBObjectList` containing all the elements in the model (represented using `EDBObject` instances) that are instances of the type.

  MQT Translation: returned `EDBObjectList` is able to query from the database model elements that are instances of the type (`SELECT * FROM OBJECT WHERE (OBJECT.CLASSID = getClassID(type))`) and the query is not executed until the result is required. The process to get the `EDBObjectList` is similar to the expression `allOfKind()`. In this case, the `getAllOfType(type)` method of the `EDBModel` instance is executed. In this method, first, `classId` of the type is obtained using EDBCache instance. Then `EDBObjectList` is instantiated. Some parameters which are passed to the constructor are: `tables = ''[Object]''`, `conditions=''(ClassID=?)''` `parameters=[getClassID(type)]`, `isOne=false`.

4.2 Element Filtering

A previously instantiated `EDBObjectList` is the input of EOL expressions of this group. The starting point of the translation is the execution of a method of the `EDBObjectList` instance, and it is performed by the *EOL Module*. The second step in Figure 3 illustrates translation of a select, an expression of this type. Next, translation process of some EOL expressions of this type is described:

- **`select(iterator:Type | condition):EDBObjectList`**:

  EOL Description: getting an `EDBObjectList` as input, it returns all elements of the list satisfying the condition.

  MQT Translation: returned `EDBObjectList` is able to query from the database model elements of the input `EDBObjectList` satisfying specified condition `(SELECT * FROM OBJECT WHERE (L2_SUBQUERY) AND (CLASSID=? OR ...))`. The query is not executed until the result is required.

  Step 2 of Figure 3 illustrates the translation process and it is explained below: First, the *EOL Module* executes the `getAbstractOperation("select")` method of the corresponding `EDBObjectList` (previously instantiated on the translation of another expression; e.g. *L1* of the figure). Then, the `execute()` method of the `EDBObjectListSelectOperation` class instance (*S1*) is executed. It returns a new instance of the `EDBObjectList` class (*L2*). In this case, constructor parameters are completed with attributes of L1 and with condition of the select: `tables = L1.tables, conditions=select_subquery + L1.conditions` `parameters=[select_parameters + L1.parameters], isOne=false`. With this information, the `EDBObjectList` instance is able to construct the SQL expression that gets model elements that meet previous conditions and also the condition of the select.

- **`reject(iterator:Type | condition):EDBObjectList`**:

  EOL Description: getting an `EDBObjectList` as input, it returns all elements of the list that do not satisfy the condition.

  MQT Translation: returned `EDBObjectList` is able to query from the database model elements of the input `EDBObjectList` that do not satisfy the condition `(SELECT * FROM OBJECT WHERE (OBJECT NOT IN L2_SUBQUERY) AND (CLASSID=? OR ...))`. When this expression is parsed, as with the select expression, the *EOL Module* executes the `getAbstractOperation("reject")` method of the corresponding `EDBObjectList`. It uses a `EDBObjectListRejectOperation` instance (based on a `EDBObjectListSelectOperation`) to instantiate the returned `EDBObjectList`.

- **`collect(iterator:Type | condition):EDBObjectList | PrimitiveValueList`**:

  EOL Description: getting an `EDBObjectList` as input, it returns a `EDBObjectList` or a `PrimitiveValueList`. The returned list contains values of the specified condition for each element of the input list.

  MQT Translation: if the specified condition is based on an attribute, the returned `PrimitiveValueList` contains values of the feature for each element of the input `EDBObjectList`. Else, if the feature is based on a reference, an `EDBObjectList` is returned.

  When this expression is parsed, the *EOL Module* executes the `getAbstractOperation("collect")` method of the corresponding `EDBObjectList`. To return the computed list, first, SQL queries should be executed and then results obtained from the database are used to populate returned list. The executed query is based on the input `EDBObjectList`, but it is re-factorized to return in one execution all the required values. If the result is a `PrimitiveValueList`, format of the SQL expression constructed and executed during collect translation is the following: `SELECT VALUE FROM OBJECT INNER JOIN ATTRIBUTEVALUE ON OBJECT.OBJECTID = ATTRIBUTEVALUE.OBJECTID WHERE ....` Else if the result is an `EDBObjectList`, the format of the SQL expression is: `SELECT VALUE, CLASSID FROM OBJECT INNER JOIN REFERENCEVALUE ON OBJECT.OBJECTID = REFERENCEVALUE.OBJECTID WHERE ...`

- **`selectOne(...), closure(...), aggregate(...), exists(...), etc.`**: The approach makes use of the previously described methods' to get the result of these methods.

4.3 Query Results (EDBObjectList)

These EOL query expressions return a result from a given `EDBObjectList`. The translation starting point is a method of the `EDBObjectList` instance that is executed by the *EOL Module*. It is important to note that as this type expressions require to return the result, SQL expression constructed by the input `EDBObjectList` should be executed. The SQL query expression is only executed once, since in the following execution of methods of the same `EDBObjectList` instance same ResultSet is used. Then, based on the `ResultSet` returned by the database, required result is prepared and returned. Third step of the Figure 3 illustrates translation of an expression of this type: at(2) that returns the second `EDBObject` of the list.

Next, translation process of some EOL expressions of this type are described:

- **`at(index: Integer):EDBObject`**:

  EOL Description: returns the `EDBObject` of the collection at the specified index.

  MQT Translation: returns a new instance of EDBObject based on the result located in the `ResultSet` (returned by the database) at the specified index. As step 3 of the Figure 3 illustrates, the *EOL Module* executes the method `get(2)` of the `EDBObjectList` (L2). To get the result, first the SQL expression constructed by the *L2* is executed through the `EDBConnection` instance (*Con*) located at the `EDBModel` instance (*Model*). The database returns a `ResultSet` that is stored on *L2*. Finally, it is used to return results, in this case creating a new `EDBObject` instance based on the second row of the `ResultSet`.

- **`first(), second(), third(), fourth(), last() : EDBObject`**:

  EOL Description: returns the first, second, third, fourth and last `EDBObject` of the `EDBObjectList` respectively.

  MQT Translation: these expressions make use of the previously described method `at(index)` (e.g. at(0) in case of first()).

- **`size():Integer`**:

  EOL Description: returns how many `EDBObjects` are contained in the `EDBObjectList`.

  MQT Translation: first, `ResultSet` is obtained (as previously described) and then returns the number of rows that it contains.

- **`includes(obj: EDBObject):Boolean`**:

  EOL Description: returns true if the `EDBObjectList` includes the `EDBObject`.

  MQT Translation: the *EOL Module* executes the `contains(obj)` method of the `EDBObjectList`. This method iterates all the rows of the previously obtained `ResultSet` analysing if the specified obj is in the list. *ObjectID* is used to compare `EDBObjects` and if it is equal once, the method returns *true*.

- **`includesAll(objCol: Collection):Boolean`**:

  EOL Description: returns true if the `EDBObjectList` includes all the `EDBObjects` of objCol.

  MQT Translation: the *EOL Module* executes the `containsAll(col)` method of the `EDBObjectList`. This method iterates all `EDBObjects` of col, executing for each one the previously described `includes(obj)`.

4.4 Query Results (EDBObject)

These EOL query expressions return a result from a given `EDBObject`. The translation starting point is a method of the `EDBObject` instance that is executed by the *EOL Module*. Next, the translation process of some EOL expressions of this type is described:

- **`.feature`** **(e.g. *obj.name*)**:

  EOL Description: returns the value of the specified feature of the given `EDBObject`.

  MQT Translation: constructs and executes the SQL expression getting the features' value from the database. In case the feature is an attribute the SQL expression has the following format: `SELECT FROM ATTRIBUTEVALUE WHERE OBJECTID=? AND FEATUREID=?`; else, if the feature is a reference, the SQL expression format is: `SELECT FROM REFERENCEVALUE WHERE OBJECTID=? AND FEATUREID=?`. As shown on the fourth step of 3, to translate the EOL expression to SQL, first, the *EOL Module* executes the `getValue(feature)`. Then, the SQL expression is constructed and executed within this method. As previously shown, the SQL query uses the *OBJECTID* and *FEATUREID*. While the *OBJECTID* is directly obtained from the `EDBObject` instance, the *FEATUREID* is obtained through the `EDBCache` instance (cache) of the *Model*. Finally, the SQL query is executed and result is returned.

- **`type(): Type`**:
  EOL Description: returns the type of the `EDBObject`.
  MQT Translation: the *EOL Module* executes the `getEClass()` method that returns `EClass` of the `EDBObject`. The `EClass` is obtained directly from the `EDBObject` instance (it is specified in the instantiation through the constructor).

5 EMPIRICAL STUDY

This section presents an empirical study of MQT. We have executed a query over five models (M1...M5) of different sizes. We have evaluated storage size, query execution time and memory footprint. The query is based on the *GraBaTs'09 Reverse Engineering Contest*, and identifies singletons within a Java project. The query has been implemented using EOL, and it is illustrated in Listing 1. First all the methods named `getInstance` that have a `modifier` are selected (lines 1-2). Then, for each method, if its `modifier` is static and public (line 7) and the return type is the same as the class that contains the method (lines 8-9), the class is stored in a variable (line 10). Finally, the query shows the name of all the classes that are singletons (lines 13-14).

```
1  var method = MethodDeclaration.all.
    select(m | m.name= 'getInstance');
2  method = method.select(m | ! m.
    modifier.isEmpty());
3  var mod;
4  var singletons = new Sequence;
5  for(m in method){
6    mod = m.modifier;
7    if(mod.static = true and mod.
    visibility.literal == 'public'){
8      var class = m.
    abstractTypeDeclaration;
9      if(m.returnType.type = class)
10        singletons.add(class);
11   }
12 }
13 for(c in singletons)
14   c.name.println();
```

Listing 1: EOL query that extracts singleton classes from a Java model.

Models have been created from existing Java plugins using the *Java Discoverer* provided by MoDisco (Bruneliere et al., 2010), a framework for model-driven reverse engineering. These models have been persisted in the native XMI and also in a single-file, embedded database using H2 v.1.3.168 and a metamodel-schema (previously described in Section 3.1). Model to database importation has been performed using the XMI to database import utility provided by MQT. Table 1 illustrates details about models used on this study: size of the XMI file, size of the database file, number of objects existing within the database, number of instances of the `MethodDeclaration` class, number of instances that satisfy first conditions (lines 1-2 of 1) and number of returned singletons.

The query has been executed 100 times over each model and using both persistence formats (XMI and embedded DB). Values measured have been storage size, memory footprint and query execution time. The memory footprint has been measured on both cases using VisualVM[2]. Regarding query execution times, measured values are different depending on the used persistence:

- XMI: the model is first completely loaded into memory and then the query is executed. As such, we have divided results in two groups: (i) first execution which includes loading time and query execution time; and (ii) the average of the following executions which include only query execution time.
- Embedded DB: the first execution has a warm-up time overhead which includes loading the database driver, connections, etc. For this reason, we have divided results in two groups: (i) first execution which includes warm-up time and query execution time; and (ii) the average of subsequent executions which include only query execution time.

All tests have been executed under an Intel Core i7-3520M CPU at 2.90GHz with 8GB of physical RAM running 64 bits Windows 7 SP1, JVM 1.8.0 and the Eclipse Luna (4.4.0) distribution configured with 2GB of maximum heap size.

5.1 Results

This section shows and discusses obtained results: storage size, execution time and memory footprint. Storage size and memory footprint are shown in Megabytes (MB) and query execution time in milliseconds (ms).

5.1.1 Storage Size

As is illustrated on Figure 4 models persisted using our approach are between %31-%53 bigger than models persisted using XMI. The reason for it is that besides model information, the database stores indexes

[2]Read more about VisualVM at http://visualvm.java.net

Table 1: Details about used models.

	XMI Size (MB)	DB Size (MB)	# Object	# MethodDecl.	# Methods cond	# Singletons
M1	45,3	59,5	165.741	5.366	9	9
M2	82,2	126	330.761	8.129	8	8
M3	212	299	875.988	11.393	6	6
M4	327	452	1.343.207	15.386	3	0
M5	403	519	1.566.890	19.366	0	0

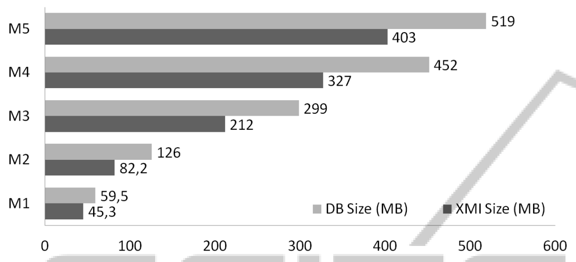
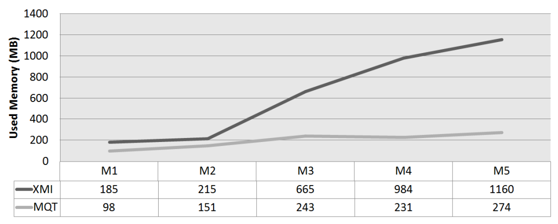


Figure 4: Comparison of storage size (MB) between XMI and Embedded DB.

and duplicated information in some cases (e.g. *FeatureValueCount* table) to support faster queries.

5.1.2 Memory Footprint



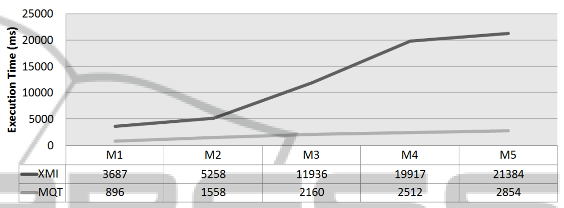
	M1	M2	M3	M4	M5
XMI	185	215	665	984	1160
MQT	98	151	243	231	274

Figure 5: Comparison of used memory (MB) between XMI and MQT.

Figure 5 shows the results of the used memory during the execution of the query over the selected models (from the smallest M1 to the largest M5). As illustrated in the figure, in XMI, the use of memory grows slowly from M1 to M2 (% 16 more) but the size of the model has much more impact over memory footprint from M3 to M5. In the first executions the used memory has a low value (185 and 215 MB), but in case of M4-M5 the used of memory rounds one gigabyte, making handling large XMI models impractical in devices with limited memory such as an embedded system, smartphones/tablets or computers with reduced memory.

In MQT, the memory usage increases, but the growth is slower: comparing with M1, memory usage is 2.79 times higher in M5, while in the case of XMI the use of memory is 6.27 bigger. Being so, although model size has impact over used memory in MQT, it scales better than XMI.

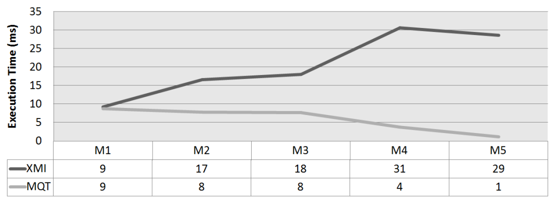
5.1.3 Query Execution Time



	M1	M2	M3	M4	M5
XMI	3687	5258	11936	19917	21384
MQT	896	1558	2160	2512	2854

Figure 6: Comparison of query execution time (ms) between XMI (with loading time) and MQT (with warm-up time).

On the one hand, Figure 6 illustrates query execution times on different models using XMI and MQT. Measures shown on this chart also include the time for loading model in case of XMI and the time for warming-up the database in case of MQT. As is illustrated in the figure, XMI has to load the entire model in memory and it is a time consuming task. By contrast, MQT has to warm-up the database (create driver and connections, load some information in the memory, etc.) but the impact of the model size is lower than in XMI.



	M1	M2	M3	M4	M5
XMI	9	17	18	31	29
MQT	9	8	8	4	1

Figure 7: Comparison of query execution time (ms) between XMI (without loading time) and MQT (without warm-up time).

On the other hand, Figure 7 illustrates average of the query execution without load/warm-up time. As shown on the chart, model size has impact over the query execution time over XMI. However, execution time in M5 is lower than execution time in M4. The reason is that as is shown on Table 1, while in M4 three methods satisfy conditions of the query program, there are not methods satisfying them in M5. Consequently, fewer queries should be translated and executed in the case of M5, and this has a positive impact over the execution time (less time required).

In case of MQT, the number of queries to be translated and executed has direct impact over the execution time result and it is dependent on the nature of the model. However, in this case, model size has not significant impact over the result.

5.2 Threats to Validity

Used memory and execution time results, show that our approach is promising in terms of scalability with respect to XMI. Correctness of the results has been verified manually, ensuring that query results are the same in XMI and in MQT.

Results show that a more intensive evaluation should be performed, analysing the impact of the nature of the model and of the query program over the execution time and memory footprint. We plan to perform this study in future iterations of this work.

6 RELATED WORK

Morsa (Pagán et al., 2013), Neo4EMF (Benelallam et al., 2014), EMF Fragments (Scheidgen, 2013), Mongo EMF (Bryan Hunt, 2014) and CDO (Eike Stepper, 2014) are approaches that provide model persistence facilities by leveraging relational and non-relational Database Management Systems (DBMSs). These approaches provide persistence-level query languages that leverage capabilities of these databases (E.G. MorsaQL (Pagán and Molina, 2014), COCL (Eike Stepper, 2014), CYPHER or SQL). Models persisted using this approach also can be queried using a model-level query language (E.G. EMF Query (Anthony Hunter, 2014), INCQuery (Bergmann et al., 2012), EOL (Kolovos et al., 2014) or OCL), but without fully-leveraging database capabilities. By contrast, our approach translates queries from model-level to persistence-level and then executes them, providing in this way a mechanisms for querying from model-level but leveraging also persistence capabilities.

Other approaches are focused on the generation of queries based on OCL-like languages: (Heidenreich et al., 2008) proposes an approach focused on generating SQL queries from invariants that are specified using OCL. This approach supports mapping between Unified Modelling Language (UML) models and databases. Then, the approach generates queries that allow to evaluate invariants using SQL. (Demuth et al., 2001) describes an approach that generates views using OCL constraints. Then it uses views to check the integrity of the persisted data. The approach has been implemented in OCL2SQL[3], a tool that generates SQL queries from OCL constraints. (Marder et al., 1999) proposes another similar approach for integrity checking. While previously described approaches translate OCL constraints into SQL queries at compilation-time, our approach translates queries from EOL to SQL, but at runtime.

The approach described in (Parreiras, 2012), translates SPARQLAS (an SPARQL-like query syntax) to SPARQL and then executes translated queries against an OWL knowledge base. Obtained results are used as input for OCL queries. Being so, this approach executes queries in persistence-level (SPARQL) and then results are the input of model-level queries (OCL). By contrast, our approach translates queries from model-level (EOL) to persistence-level (SQL) and then executes them against the database.

(Kolovos et al., 2013) describes an approach where EOL is used to query large datasets stored on relational databases composed by one table. This approach uses the naive translation provided by EMC to query information persisted in a single-table database. By contrast, our approach provides custom translation of SQL queries to be executed against a database with multiple tables.

7 CONCLUSIONS AND FURTHER WORK

In this paper, we have presented MQT, a prototype that is able to query models persisted in a relational database at model-level (same level of abstraction used for querying models persisted with XMI) but exploring the advantages of a persistence-level language (SQL). While existing approaches which are able to translate OCL-like languages statically at compilation time, our approach provides runtime translation of queries from EOL to SQL. Performing the translation at runtime eases to translate languages that have not direct mapping, and this is the case of EOL and SQL. At this stage, MQT only supports read-only EOL queries. However, we plan to add support for modification queries in a future prototype.

We have performed an empirical study where querying using both MQT and XMI is compared. In this study, we have compared the values that measure storage size, memory footprint and query execution time. The results show that our approach scales better than XMI on the execution of the query proposed by

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

the *GraBaTs'09 Reverse Engineering Contest*. While model size has a substantial impact on the memory footprint and loading time using XMI, the impact is softer in the case of MQT. However, results show that number of queries to be translated and executed has direct impact over the execution time. The number of queries is dependent on the nature of the model.

For future work, we plan to perform a complete study that will involve: (i) experimenting with different database configurations and with different in-memory caching strategies; (ii) experimenting with queries of different types (e.g. more complex, returning more results, etc.); (iii) experimenting with models of different sizes returning similar results and with models of same size returning more results; and (iv) comparing results with other persistence approaches.

Open issues for the future are: (a) implementation of the `Resource` interface of EMF to integrate the approach with EMF-based tools; and (b) analysis of whether the approach could be generalized to support additional persistence approaches and querying languages.

ACKNOWLEDGEMENTS

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

REFERENCES

Anthony Hunter (2014). EMF Query. http://projects.eclipse.org/projects/modeling.emf.query. Accessed June 13, 2014.

Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., and Launay, D. (2014). Neo4EMF, A Scalable Persistence Layer for EMF Models. In Cabot, J. and Rubin, J., editors, *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer International Publishing.

Bergmann, G., Hegedüs, A., Horváth, A., Ráth, I., Ujhelyi, Z., and Varró, D. (2012). Integrating efficient model queries in state-of-the-art emf tools. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 1–8, Berlin, Heidelberg. Springer-Verlag.

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). 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, pages 173–174, New York, NY, USA. ACM.

Bryan Hunt (2014). Mongo EMF Wiki. https://github.com/BryanHunt/mongo-emf/wiki. Accessed March 17, 2014.

De Carlos, X., Sagardui, G., and Trujillo, S. (2014). MQT, an Approach for Runtime Query Translation: From EOL to SQL. In *Proceedings of the 14th International Workshop on OCL and Textual Modeling Applications and Case Studies*, OCL '14.

Demuth, B., Hussmann, H., and Loecher, S. (2001). Ocl as a specification language for business rules in database applications. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML '01, pages 104–117, London, UK, UK. Springer-Verlag.

Eike Stepper (2014). CDO Model Repository Overview. http://www.eclipse.org/cdo/documentation/. Accessed March 17, 2014.

Heidenreich, F., Wende, C., and Demuth, B. (2008). A framework for generating query language code from ocl invariants. *ECEASST*, 9.

Kolovos, D., Rose, L., Garca-Domnguez, A., and Paige, R., editors (2014). *The Epsilon Book*. Enterprise Systems, University of York.

Kolovos, D. S., Wei, R., and Barmpis, K. (2013). An Approach for Efficient Querying of Large Relational Datasets with OCL-based Languages. In *XM 2013– Extreme Modeling Workshop*, page 48.

Marder, U., Ritter, N., and Steiert, H. (1999). A dbms-based approach for automatic checking of ocl constraints. In *Proceedings of Rigourous Modeling and Analysis with the UML: Challenges and Limitations*, OOPSLA.

Pagán, J. E., Cuadrado, J. S., and Molina, J. G. (2013). A Repository for Scalable Model Management. *Software & Systems Modeling*, pages 1–21.

Pagán, J. E. and Molina, J. G. (2014). Querying large models efficiently. *Information and Software Technology*, 56(6):586 – 622.

Parreiras, F. S. (2012). *Semantic Web and Model-driven Engineering*. John Wiley & Sons.

Scheidgen, M. (2013). Reference representation techniques for large models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 5:1–5:9, New York, NY, USA. ACM.