Supporting CRUD Model Operations 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-Mondragon, Spain*
[2]*Mondragon Unibertsitatea, Goiru 2, 20500 Arrasate-Mondragon, Spain*

Keywords: Model Driven Development, Persistence, Model Queries, Model Operations, Database.

Abstract: Model-based software development promises improvements in terms of quality and cost by raising the abstraction level of the development from code to models, but also requires mature techniques and tools. Although Eclipse Modelling Framework (EMF) introduces a default persistence mechanism for models, namely XMI, its usage is often limited as model size increases. To overcome this limitation, during the last years alternative persistence mechanisms have been proposed in order to store models in RDBMS and NoSQL databases. Under this new paradigm, model operations can be performed at model-level and persistence-level, e.g., mapping EOL model operations into SQL statements. In this paper, we extend our framework (called MQT) to support CRUD (Create, Read, Update and Delete) operations from model- (EOL) to persistence-level (SQL), using a streaming execution of queries at run-time. Through comparable evaluation metrics, we evaluate the performance and memory footprint of the framework using the GraBaTs scenario.

1 INTRODUCTION

Model Driven Development (MDD) intended to raise the abstraction level from code to models, using models as first-class citizens of the development process (Atkinson and Kuhne, 2003). The MDD adoption encompasses additional tasks such as model validation, constraint checking or model analysis, often requiring model querying and modification. As in the traditional software development process, models may become very large (up to millions elements), so scalable model persistence and operation mechanisms are highly demanded (Gómez et al., 2015).

Eclipse Modelling Framework (EMF)[1] is a mature and a widely used modelling framework provided within the Eclipse IDE, which adopts XML Meta-data Interchange (XMI)[2] as the default mechanism to persist models. However, as the size of the model increases, XMI potentially may entail memory and computation problems (Benelallam et al., 2014; Espinazo Pagán and García Molina, 2014). Consequently, over the last decade, additional persistence mechanisms have been proposed to effectively support large-scale models containing a bunch of elements such us in reverse engineering, construction or wind-power (Bagnato et al., 2014)). These recent approaches opt for the use of databases as a persistence mechanism (e.g. Morsa (Pagán et al., 2013), or Neo4Emf (Benelallam et al., 2014)).

When models are persisted in a database, engineers may operate with models in two different ways: at **model-level** and **persistence-level** (also referred to as database-level). *Model-level* operations are closer to modelling engineers due to the modelling language nature. Hence, if a persistence mechanism is changed or evolved, model operations still remain valid. Some examples of model-level languages are: Epsilon Object Language (EOL) (Kolovos et al., 2010), Object Constraint Language (OCL)[3], EMF Query (Hunter, 2015), and IncQuery (Ujhelyi et al., 2015). *Persistence-level* operations are database dependant. This type of operations are typically executed directly over the database, with an inherent straightforward performance optimization. Examples here include: SQL for relational databases or MorsaQL (Espinazo Pagán and García Molina, 2014), and Morsa for models (Pagán et al., 2013).

In a previous work, we presented the first bits of MQT (Model Query Translator), in a preliminary attempt to translate a model-level imperative language (EOL) into a persistence-level declarative language (SQL) at run time. In this paper, we present a proof-of-concept prototype that supports translation

[1]https://eclipse.org/modeling/emf/
[2]http://www.omg.org/spec/XMI/
[3]http://www.omg.org/spec/OCL/

Carlos, X., Sagardui, G. and Trujillo, S.
Supporting CRUD Model Operations from EOL to SQL.
DOI: 10.5220/0005644401530160
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 153-160
ISBN: 978-989-758-168-7

```
1 EClass.all.printAbstractNames();
2 for (c in EClass.all){
3    c.eSuperTypes.
   printAbstractNames();
4 }
5 operation Collection<EClassifier>
   printAbstractNames(){
6   self.select(c | c.abstract).name
   .println(); }
```

Listing 1: Sample model-level operation specified using in EOL.

of CRUD (Create, Read, Update, Delete) operations from EOL to SQL using streaming execution at runtime. To the best of our knowledge, there is not any other framework that supports CRUD operations from EOL model operations to SQL.

2 TOWARDS RUN-TIME QUERIES IN MODELS

Epsilon Object Language (EOL) (Kolovos et al., 2006) is an imperative programming language for creating, querying and modifying EMF models. It is a mixture of Javascript and OCL, and facilitates specification of queries and model modifications. For instance, Listing 1 illustrates a model-level query specified using EOL. In this example, the query prints the names of all abstract `EClass` instances (line 1) and uses the `printAbstractNames` (lines 5 to 7) operation for this purpose. All `EClass` instances are iterated (lines 2 to 4), and the names of each abstract super-type are printed (line 3). Such query statement also employs the `printAbstractNames` operation for printing the names of the abstract candidates only.

SQL is a structured and mature language to query or modify information from RDBMS (and also from some NoSQL databases such as MongoDB[4] or Neo4J[5]). While EOL is an imperative language, SQL [6] is declarative. If we translate an EOL query into a declarative language (e.g. SQL), the `printAbstractNames` operation from Listing 1 should be translated differently for the statement in line 1 and for the one in line 3.

For Listing 1, if translation is performed at compilation-time, it will be necessary to decide query mapping beforehand. Such static query-analysis is avoided in case of run-time translation, since it will allow the system to adapt the target SQL query for each particular scenario at execution time.

Moreover, there also exist some constructs that have not direct mapping between EOL (imperative) and SQL (declarative). For example, EOL supports loop statements (line 5 in Listing 1) that are not supported, at least natively, in SQL. This means that in compilation-time translation, such construct mismatch increases the inherent translation complexity, so run-time translation is more appropriate for flexible mapping (Carlos et al., 2015).

3 CRUD OPERATIONS IN MQT

In a first version, the MQT framework was able to partially translate EOL operations into SQL and operate with `SELECT` operations against a H2 embedded database (Carlos et al., 2015). In this paper, we extend the MQT framework which supports: the *streaming execution of queries*, and the translation process for all *CRUD operations*.

Figure 1a illustrates an overview of our framework. As depicted in the figure, the EOL Module is the responsible for parsing and executing model operations. The EMC (Epsilon Model Connectivity Layer) provides different interfaces to connect and communicate with the EOL Module. In this sense, MQT implements such interfaces for EOL integration.

The domain metamodel provides valuable information during the translation process. MQT uses this information to optimize SQL queries, as well as to perform correctness checking for query results:

- *Feature Cardinality* is used to optimize query performance by adding a limit clause (e.g. SELECT * FROM ... LIMIT N) into the translated SQL. Feature cardinality also checks if a query returns the expected numbers of results.
- *Feature Type* for checking if a query returns instances of the expected type.
- Subtypes. Know which classes extend a specific `EClass` instance. This is useful when all instances of a specific type are queried.

MQT uses a metamodel agnostic data-schema to persist models (see Figure 1(b). For more information about the persistence approach please refer to our previous work (Carlos et al., 2015).

3.1 Streaming Execution of Queries

MQT provides streaming execution of translated SQL statements. This means that queries are only executed

[4]http://www.unityjdbc.com/mongojdbc
[5]https://goo.gl/l22XZO
[6]http://www.w3schools.com/sql/default.asp

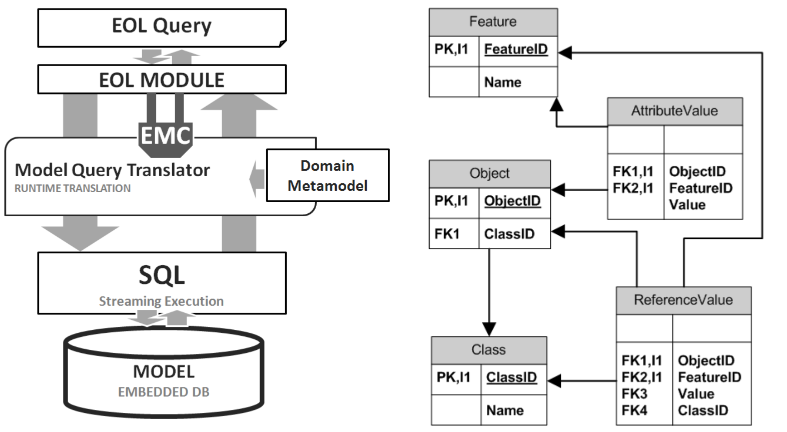

Figure 1: (a) Overview of the framework and (b) database-schema used in MQT.

on-demand, with two major benefits: (i) a *lower memory usage* since the information is queried and maintained in memory only when needed, and (ii) *avoids partial results* which increase memory usage and execution speed.

This is achieved through lazy-objects, with two different implementations:

1. `EDBObject`. Each instance of this class represents a model-object. Each `EDBObject` instance contains the ID of the model element within the database and the type of the object. This lazy-object is able to construct SQL statements in order to obtain database attributes and reference values of the represented model element. Values are only obtained from the database when needed.
2. `EDBCollection`. This class is employed to instantiate collections of model elements or feature values. Each instance is a lazy-collection that is able to construct a SQL statement based on translation information. The statement is only constructed and executed when the information of the list is required by another operation. Additionally, `EDBCollection` instances are able to customize the constructed SQL statement to obtain the information from the database more efficiently (e.g. the SQL statement would not be the same when querying the size of a list or all elements within a list).

During the translation process, both implementations, i.e. `EDBObject` or `EDBCollection`, are instantiated by the `EDBModel`. This latter class is the responsible for constructing and executing SQL statements with the aim of creating new model element instances. Additionally, the `EDBModel` class is the starting point for the streaming translation and execution of the SQL statements.

Figure 2 illustrates the streaming construction and execution of a SQL query from a EOL operation. The operation prints the name of the first abstract `EClass` instance specified within the queried model. Steps of the translation are detailed below:

1. The `EDBModel` creates a new instance of a lazy-collection containing model elements (`L1`). This collection obtains information from the translated query (at this point that objects should be of type `EClass`). However, the SQL statement is not constructed nor executed since the information is not required at this point.
2. The `EDBModel` creates another lazy-collection (`L2`). This instance inherits information of `L1` and also obtains new information related with the condition. At this point the `L2` collection is able to construct a SQL statement which obtains all `EClass` instances that are abstract. However, it is not constructed at this point since the results are not required.
3. MQT parses and translates the `first()` EOL operation in the third step. This step requires to obtain the first element of the collection. With this purpose, `L2` collection constructs an SQL statement with the information provided in the previous steps. Then executes the SQL query against the database. The SQL statement returns all abstract `EClass` instances to the `L2` collection. Next, `L2` gets the first result returned by the database and instantiates an `EDBObject` specifying it(`OBJ1`). `L2` returns `OBJ1` to the `EOL Module`.
4. The `name` of the previously returned element is required, since it has to be printed in the screen. Being so, the `OBJ1` instance constructs and executes an SQL statement which obtains the value of the `name` attribute in the database. Finally, the value is returned and printed.

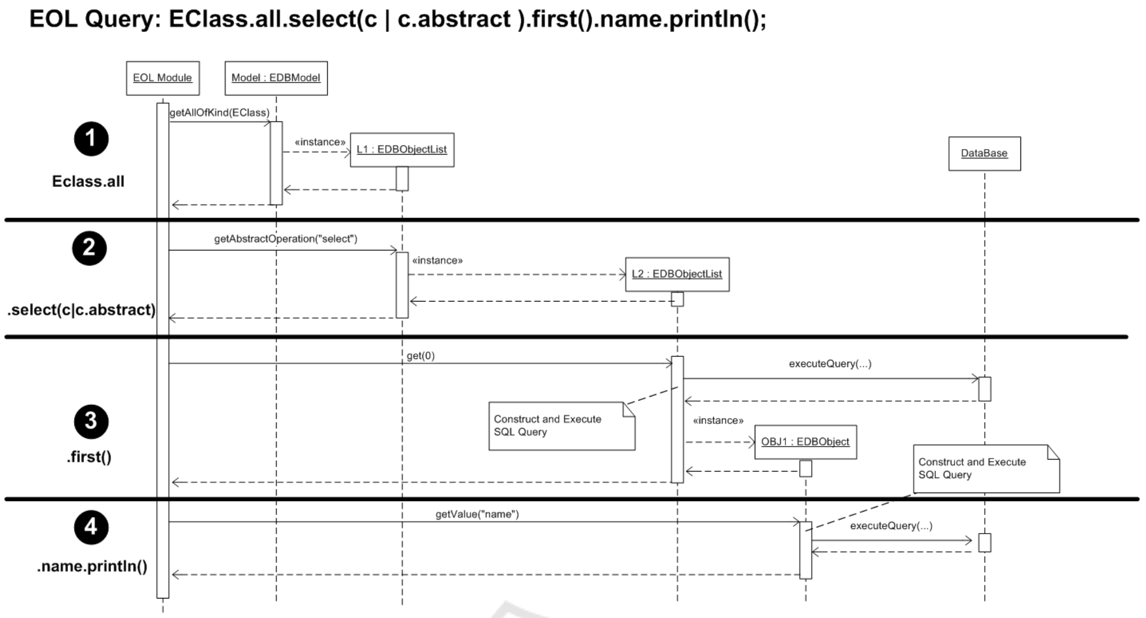

Figure 2: Run-time translation process and streaming execution.

3.2 Support for Model-Level CRUD Operations

Table 1 illustrates the process followed by MQT for translating and executing CRUD model operations from EOL to SQL. Is important to note that with MQT all the translated and executed SQL statements within a same operation are committed only when the operation is executed completely. If commits are performed after executing each translated SQL statement, and the operation is not executed completely, result can be a corrupted model.

4 EMPIRICAL STUDY

In this section, we aim to provide memory and performance metrics for the run-time query translation for different model CRUD operations and compare it with XMI. The EOL operations have been executed using MQT and XMI. Results correctness has been automatically validated by comparing query results of both approaches, i.e. MQT and XMI. In order to get reliable numbers, each query was processed 10 times for each case and discussed the results against the following quantitative **metrics**:

- *Execution Time Average:* Execution time average of each case expressed in milliseconds (ms).
- *Memory Usage:* Maximum memory usage during query execution for each case expressed in megabytes (in MB).

In MQT, execution time includes both time for query translation and time for query execution. In XMI, execution time includes time for loading the model in memory, time for executing the query and time for persisting modifications. In the case of the Read type operation, we also provide the query execution time required by XMI when the model has been previously loaded in memory (not possible in CREATE, Update and Delete operations).

The experiment was executed as a standalone application on a Intel Core I7-3520M @ 2.90 GHz, 8GB, Windows 7 SP1 (64-Bit) operative system and Java SE v1.8.0. We have repeated 10 times the execution of each model operation and in each execution the virtual machine was restarted. We provide metrics for models of the GraBaTs 2009 scenario (Sottet et al., 2009), since it is widely used to evaluate model persistence and querying approaches. These models specify source code of different Java packages and conform to the JDTAST metamodel which contains abstractions of the Java source code. Table 2 summarizes different properties of the GraBaTs' models (**Set0-4**). We have obtained evaluation metrics for the following **model CRUD operations**:

- **OP1: Create Operation.** Creates one `Class` per each existing `TypeDeclaration` (see Listing 2). The name of the `Class` instance will correspond with the name of the `TypeDeclaration` instance.
- **OP2: Read Operation.** Query from the GraBaTs case study (Sottet et al., 2009) that prints the names of all the singleton classes (Listing 3).

Table 1: Translation process of different CRUD operations.

Operation	EOL Operation	MQT Translation	Generated SQL queries
		Operations with model elements	
Create model element	new EClass	`EDBModel.createInstance()` constructs and executes a SQL insert that adds a new element within the database.	`INSERT INTO Object`

Delete model element	—	`EDBModel.deleteElement()` constructs and executes SQL statements that delete: the element itself, the attributes and references of the element, and childrens of the element.	`DELETE FROM Object`
			`DELETE FROM AttributeValue`
			`SELECT refElements FROM ReferenceValue`
			`foreach(elem in refElements)`
			`EDBModel.deleteElement(object)`

		Operations with single-value features	
Update attribute or non-containment reference	c.name="E1"	`EDBObject.setValue()` constructs a SQL update that replaces old value with the new one.	`UPDATE AttributeValue / ReferenceValue`

Update containment reference	c.children = elem	`EDBModel.setValue()` method first checks if the reference contains a value (SQL select) and if contains it is deleted with a SQL delete. Then, another SQL select checks if the new reference is contained by other element. If is contained, a SQL update is generated and executed, else an insert SQL is generated and executed.	`SELECT oldValue FROM ReferenceValue`
			`if(oldValue != null)`
			`EDBModel.deleteElement(oldValue)`
			`SELECT parent FROM ReferenceValue`
			`if(parent != null)`
			`UPDATE ReferenceValue`
			`else`
			`INSERT INTO ReferenceValue`

		Operations with multi-value features	

Create value	c.eSuperFeatures.add(newElem)
	c.eSuperFeatures.addAll(elemList)
	...
		`EDBCollection.add()`, `EDBCollection.addAll()` all call to `EDBCollection.addValue()` that first checks with a SQL select if the new value can be added. If it can be added, an SQL insert is constructed and executed for adding the new value.	`featureNumber = SELECT COUNT(features)`
			`FROM AttributeValues/ReferenceValues`
			`EDBModel.deleteElement(oldValue)`
			`EDBModel.deleteElement(oldValue)`
			`if featureNumber < maxValues`
			`INSERT INTO AttributeValues/ReferenceValues`
			`else`
			`showErrorAndTerminate()`

Update value	c.eSuperFeatures = refList	`EDBCollection.setValue()` checks if the feature contains values. If contains, it calls EDBCollection.deleteElement() to delete them. Finally it calls EDBCollection.addValue() to set the new values.	`SELECT oldVal FROM AttributeValues/ReferenceValue`
			`EDBCollection.removeAll(oldVal)`
			`EDBCollection.addAll(newVal)`

Delete value	c.eSuperFeatures.remove(elem)
	c.eSuperFeatures.clear()
	...
		`EDBCollection.remove()` `EDBCollection.removeAll()` all call to `EDBCollection.remove()`. This method removes feature value from the model element. If the reference is of containment type, contained elements are recursively deleted.	`if feature.isContainmentReference()`
			`EDBModel.deleteElement(elem)`
			`else`
			`DELETE FROM AttributeValue/ReferenceValue`

Table 2: Characteristics of the GraBaTs models.

	Set0	**Set1**	**Set2**	**Set3**	**Set4**
Size (MB)	8,8	27	271	598	646
Classes	14	40	1.6K	5.7K	5.9K
Elem.	70K	198K	2M	4.8M	4.9M

- **OP3: Update Operation.** Renames all the existing `TypeDeclaration` instances (Listing 4).
- **OP4: Delete Operation.** Selects the first `CompilationUnit` instance and then deletes all the elements that are contained in the reference called `types` and all the elements contained by them (Listing 5).

```
for(type in TypeDeclaration.all){
 var class = new EClass;
 class.name = type.name.
     fullyQualifiedName;}
```

Listing 2: Create type operation.

```
var t= TypeDeclaration.all.select(
    td|
td.bodyDeclarations.exists(md:
    MethodDeclaration|
md.modifiers.exists(mod:Modifier|
    mod.public==true) and md.
    modifiers.exists(mod:Modifier|
    mod.static==true) and md.
    returnType.isTypeOf(SimpleType
    ) and md.returnType.name.
    fullyQualifiedName == td.name.
    fullyQualifiedName));
 for (iterator in t)
  iterator.name.fullyQualifiedName
      .println();
```

Listing 3: Read type operation.

4.1 Discussion

Table 3 illustrates the memory and performance results for each operation (OP1-4) for Set0-4. We have further analyzed the evaluation results below:

Table 3: Obtained results: execution time average (in milliseconds) and maximum memory usage (in Megabytes).

		SET 0		SET 1		SET 2		SET 3		SET 4	
		Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
CREATE	XMI	2626	169	6173	304	24104	1078	60551	2159	73084	2309
	MQT	98	125	171	122	1665	175	3830	355	4085	366
READ	XMI	2663	163	3980	310	26541	1127	76407	2284	81538	2437
	MQT	164	130	216	118	1646	187	3075	317	3288	316
UPDATE	XMI	4880	165	8700	328	38313	1505	100415	3069	145246	3305
	MQT	101	116	200	124	2351	181	8406	374	9532	399
DELETE	XMI	3394	155	4607	330	29446	1539	86707	3069	87736	3241
	MQT	5705	699	14278	755	80137	667	224272	449	245803	474

```
for(type in TypeDeclaration.all)
 type.name.fullyQualifiedName = "
     NewName";
```

Listing 4: Update type operation.

```
var compUnit = CompilationUnit.all
    .first();
compUnit.types.clear();
```

Listing 5: Delete type operation.

Create Operation. With this operation we have inserted 0,02% of the model elements in Set0 and Set1, 0,08% in Set2 and 0,10% in Set3 and Set4. *Execution Time:* Differences in time consumption are very relevant as MQT is between 14 and 36 times faster than XMI. Analysing the execution time for insertion with respect to the total number of elements of the models, in Set0 and Set1 while MQT takes 0,001 ms for each element in XMI it takes 0,03 ms. In bigger models, while the execution time per each element is lower in both approaches, in XMI is around 0,013 ms, MQT takes 0,0008 ms for each model element. Looking at the execution time with respect to the number of inserted model elements, MQT ranges from 7 ms for each inserted element in Set0 to 0,7 ms in Set4. In XMI it takes 187 ms for each inserted element in Set 0 and 12 ms in Set4. In both approaches, the time to insert an element decreases as the number of inserted elements grows, which indicates an overhead independent of the number of elements inserted. *Memory Usage.* XMI consumes more memory than MQT. Differences in memory consumption are bigger as the size of the models grows. In Set0, MQT consumes 26% less memory than XMI, in Set1 60% and in Set2, Set3 and Set4 consumes around 84% less than XMI. By using the advantages of operating at persistence level, memory consumption is very low in MQT (366,2 MB in the biggest model) which indicates that is more appropriate for bigger models than XMI.

Read Operation. *Execution Time.* Size of the model has a great impact over the time required to execute the read operation with XMI as the model has to be loaded in memory before operating it. Consequently, XMI requires more time than MQT to execute the query. From Set0 to Set2 XMI is between 16 and 18 times slower than MQT. But this difference is increased on the two largest models (Set3 and Set4) where MQT is 24 times faster than XMI. Additionally, if we compare the results of MQT with the time required by XMI when the model has been already loaded in memory, these are similar in both cases. *Memory Usage.* The memory usage with XMI and MQT is similar in the case of Set0 (around 150MB). In Set1, while the memory usage is duplicated respect to Set0 if XMI is used, memory usage is similar in MQT. The model size has higher impact over memory usage in Set2 with XMI and it is increased to around 1.1GB. By contrast, memory usage impact is lower if MQT is used and it only increases 69MB. In Set3, while memory usage is duplicated respect to Set2 if XMI is used (using around 2.2GB), MQT only requires 317MB. The trend is similar in Set4 where XMI requires more than 2.4GB and MQT only uses 316MB. These memory usage results show how the streaming execution of SQL statements provided by MQT minimizes the memory usage when models are queried, since it only loads the required information.

Update Operation. With this query we have updated 0,02% of the model elements in Set0 and Set1, 0,08% in Set2 and 0,10% in Set3 and Set4. *Execution Time.* XMI requires to serialize again the modified model (from memory) and MQT only requires persisting changes. Being so, MQT is 48 times faster than XMI for Set0, 43 times faster for Set1, 16 times faster for Set2, around 12 times faster for Set3 and 15 times faster for Set4. Looking at the execution time with respect to the number of updated model elements, MQT ranges from 7.2 ms for each inserted element in Set0 to 1.5 ms in Set4. XMI takes 348 ms for each inserted element in Set 0 and 24 ms in Set4. In both approaches, the time to update an element is around double time to insert an element. Update requires first to find the element and then perform the update. *Memory Usage.* XMI consumes more memory than MQT. Differences in memory consumption are bigger as the size of the models grows. In Set0, MQT consumes 30% less memory than MQT, in Set2 62% and in Set2, Set3 and Set4 consumes 88% less than XMI. While in XMI memory consumption for Set4 is 20 times bigger than Set0, for MQT is only 3.44 times bigger.

Delete Operation. *Execution Time.* In the case of MQT, performance values are not assumable for large models. The design of the delete operation translation is recursive: deleting an element also requires performing additional operations to delete children contained by the deleted element. This design derives in a translation that contains many SQL statements. There is need of re-factoring the translation for the delete operations. In the case of XMI, entire model is previously loaded in-memory and the operation for obtaining elements to be deleted, and then delete them, is resolved faster. However, XMI requires more than 86 seconds for performing delete operation on the largest models (Set3-4). *Memory Usage.* While memory usage values are higher than XMI in Set0-1 (4.5 times for Set0 and 2.3 times for Set1), memory usage is lower than XMI in the other models (2.3 times for Set2 and 6.8 for Set3-4).

4.2 Threats to Validity

We have used metamodels and models from the GraBaTs 2009 case study, a widely used case study to evaluate model persistence and querying approaches. However, a more intensive evaluation should be performed analysing the impact of the nature of the model and the metamodel over the execution time and memory footprint.

5 RELATED WORK

There are other proposals to generate persistence level queries from model-level queries: In (Egea et al., 2010), the authors describe an approach focused on generating MySQL code from a given OCL expressions. An approach to generate SQL queries from OCL invariants is presented in (Heidenreich et al., 2007). In (Demuth et al., 2001), the authors describe an approach that generates views using OCL constraints, and then uses them to check the integrity of the persisted data. The approach has been implemented using a tool that generates SQL queries from OCL constraints (OCL2SQL[7]). A similar approach for integrity checking is proposed in (Marder et al., 1999). DresdenOCL (Demuth and Wilke, 2009) provides a tool set that supports parsing and evaluating OCL constraints on various models (e.g. EMF, Java, or UML). Additionally, the approach also provdes Java/AspectJ and SQL code generation. All these previously described approaches translate OCL queries (Read type operations) into SQL. The approaches provide a compilation-time translation since both languages are declarative and they have direct mapping. By contrast, our approach translates EOL, a imperative language, to a declarative language (SQL). For this reason, our framework performs the translation at run time. Moreover, as an imperative language is translated, MQT supports translation of CRUD operations.

Another work (Parreiras, 2012) describes an approach that, first executes queries in persistence-level (SPARQL). And then uses obtained results as the input of model-level queries (OCL). While this approach translates queries from persistence-level to model-level, our approach provides translation from model- to persistence-level.

6 CONCLUSIONS

In this paper, we have presented an extension of the MQT framework to support run-time translation and streaming execution of model CRUD operations from EOL to SQL. The framework provides the model engineers the usability of operating at model-level but at the same time taking advantages of database performance. We have tested the framework for different CRUD operations (OP1-4) for models of different sizes (Set0-4). Overall, MQT seems to perform properly in terms of memory and time for create, read and update operations but not for delete. MQT requires less memory than XMI and at the same time, execution times are also better than XMI.

For future work, we will perform an evaluation taking into account more operations and larger models. We also plan to improve the translation and execution mechanism for **delete** type operations.

[7]“http://dresden-ocl.source”

ACKNOWLEDGEMENTS

This work is partially supported by the EC, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (#611125). The authors wish to thank Aitor Murguzur, Xabier Mendialdua and Dimitris Kolovos for his participation on this work.

REFERENCES

Atkinson, C. and Kuhne, T. (2003). Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41.

Bagnato, A., Brosse, E., Sadovykh, A., Maló, P., Trujillo, S., Mendialdua, X., and Carlos, X. D. (2014). Flexible and scalable modelling in the MONDO project: Industrial case studies. In *Proceedings of the 3rd Workshop on Extreme Modeling, Valencia, Spain, September 29, 2014.*, pages 42–51.

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.

Carlos, X. D., Sagardui, G., Murguzur, A., Trujillo, S., and Mendialdua, X. (2015). Model Query Translator - A Model-Level Query Approach For Large-Scale Models. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, Angers, France*.

Demuth, B., Hussmann, H., and Loecher, S. (2001). OCL as a Specification Language for Business Rules in Database Applications. In Gogolla, M. and Kobryn, C., editors, *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 104–117. Springer Berlin Heidelberg.

Demuth, B. and Wilke, C. (2009). Model and object verification by using dresden ocl. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*, pages 687–690.

Egea, M., Dania, C., and Clavel, M. (2010). MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL. *Electronic Communications of the EASST*, 36.

Espinazo Pagán, J. and García Molina, J. (2014). Querying Large Models Efficiently. *Inf. Softw. Technol.*, 56(6):586–622.

Gómez, A., Tisi, M., Sunyé, G., and Cabot, J. (2015). Map-based transparent persistence for very large models. In Egyed, A. and Schaefer, I., editors, *Fundamental Approaches to Software Engineering*, volume 9033 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg.

Heidenreich, F., Wende, C., and Demuth, B. (2007). A Framework For Generating Query Language Code From OCL Invariants. *Electronic Communications of the EASST*, 9.

Hunter, A. (2015). EMF Query. https://projects.eclipse.org/projects/modeling.emf.query. [Online; accessed 02-February-2015].

Kolovos, D., Paige, R., and Polack, F. (2006). The epsilon object language (eol). In Rensink, A. and Warmer, J., editors, *Model Driven Architecture Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg.

Kolovos, D. S., Rose, L. M., and Paige, R. F. (2010). The epsilon book (2010).

Marder, U., Ritter, N., and Steiert, H. (1999). A DBMS-Based Approach for Automatic Checking of OCL Constraints. In *Proceedings of OOPSLA*, volume 99, pages 1–5.

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

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

Sottet, J.-S., Jouault, F., et al. (2009). Program comprehension. In *Proc. 5th Int. Workshop on Graph-Based Tools*.

Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., and Varró, D. (2015). EMF-IncQuery: an Integrated Development Environment for Live Model Queries. *Sci. Comput. Program.*, 98:80–99.