Two-Step Transformation of Model Traversal EOL Queries for Large CDO Repositories

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

[1] 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. Recent approaches persist models in databases to overcome performance and memory limitations of XMI. Among them, Connected Data Objects (CDO) is a database-based model repository widely used in Model Based Engineering by academia and industry. Model traversal queries are intensively used in modelling scenarios and their performance greatly impacts tools performance and user experience. In this paper, we introduce the CDO-QT framework to transform model traversal queries from Epsilon Object Language (EOL) into SQL queries and execute them at CDO repositories. This way, model engineers can define queries using domain concepts at performance similar to SQL. We have evaluated CDO-QT executing a set of queries over repositories from 15 MB to 5 GB size. CDO-QT results in better performance and memory consumption with respect to other approaches (Plain EMF, MDT OCL, CDO-OCL).

Keywords: Model driven development · Query · Model persistence · Eclipse modelling framework · Connected data objects · Large models

1 Introduction

Model Based Engineering (MBE) raises the abstraction level of software development promising productivity increases and greatly improved quality of the code and development process [11]. In this paradigm, models automate and guide the development processes and engineers focus on domain concepts rather than on implementation details.

Modelling scenarios in industry can be really complex [1], with large models of size of 100 MB and beyond, and with millions of model elements. Engineers use modelling tools for model transformation, validation or execution. Performance might have adverse effect on development, which makes MBE adoption difficult in industry. Among all the activities, model queries are intensively used. Therefore the impact of query performance on tool performance and user experience is significant [4]. In practise, model traversal queries are the most commonly used type of queries [9]. These queries obtain all the instances of a specific type and require traversing the entire model.

P. Stevens and A. Wąsowski (Eds.): FASE 2016, LNCS 9633, pp. 141–157, 2016.
DOI: 10.1007/978-3-662-49665-7_9

The Eclipse Modelling Framework (EMF) is a mature framework widely used by the industry and academia. By default, EMF models are persisted using XML Metadata Interchange (XMI). XMI entails memory problems for models [9,15] and requires to completely load them in-memory for model traversal queries. EMF provides also a binary format which improves scalability of XMI, but it also requires loading entire models. Alternative proposals to XMI for large models choose databases for persistence, overcoming limitations by partial loading of models. Different back-end strategies have been proposed: noSQL databases (e.g. Morsa, MongoDB, NeoEMF/Map, NeoEMF/Graph or EMF Fragments); relational databases (e.g. Teneo); or several kinds of databases (e.g. CDO). Databases improve significantly performance in model operation for large models. For example, a database-based prototype introduced at [2] executes a model traversal query (GraBaTs query) 20 times faster than XMI and requires 57 % of the memory used by XMI.

Matureness and collaboration support makes Connected Data Objects (CDO) [17] the most widely used model repository in academia and industry [8]. CDO provides support for model operation from Plain EMF and EMF-based model query languages executed at *client-side* (e.g. MDT OCL or EMF query), model query languages executed at *server-side* (CDO OCL) and persistence-specific query languages (e.g. HQL and SQL). Persistence-specific languages improve significantly model operation performance. For example, GraBaTs query for CDO with H2 relational back-end for a model of almost 5 millon elements[1] requires 6 seconds and 289 MB of memory usage from SQL, while in the best case of model query languages, it requires 28 s using 525 MB. However, in model query languages model engineers use domain specific concepts, while in persistence-specific query languages engineers should be aware of the way information is persisted and learn database specific concepts and languages. This increases programming effort to get complex queries correct.

The main contribution of this paper is a framework (CDO-QT) that transforms queries from a model query language (Epsilon Object Language [12]) to a persistence specific query language (SQL) and executes them at a CDO repository. Generated queries are fully integrated with the versioning/branching of CDO. This way, model engineers can define queries using domain concepts at performance similar to SQL. CDO-QT is designed in a two-step transformation process to provide re-usability and extensibility. We evaluate performance and memory usage of different model traversal queries using Plain EMF, MDT OCL, CDO OCL, SQL and CDO-QT. We have executed the queries over ten CDO repositories from 15.3 MB to 5 GB. Results show that CDO-QT is able to execute all the queries faster and requiring less memory than the other solutions (Plain EMF, MDT OCL and CDO-OCL).

The rest of the paper is organized as follows: Sect. 2 introduces CDO and describes the motivation of this work. Section 3 describes the query transformation process performed by CDO-QT. In Sect. 4 we evaluate CDO-QT comparing performance and memory usage for executing different model traversal queries

[1] More details about the evaluation scenario and metrics in Sect. 4.

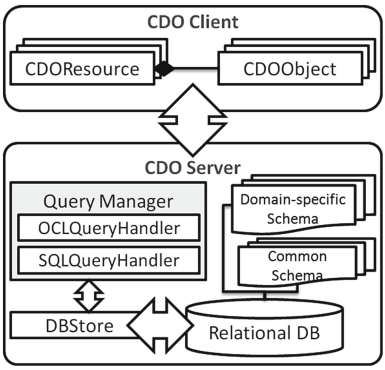


Fig. 1. Simplified CDO architecture for relational backends.

Table 1. Execution time (s) and memory usage (MB) for the GraBaTs query.

		Set0	Set1	Set2	Set3	Set4
Plain EMF	time	18	48	473	1069	1140
	mem	400	1028	3407	5672	6133
MDT OCL	time	18	46	453	1023	1101
	mem	322	934	3112	5731	6110
CDO OCL	time	1	2	11	26	28
	mem	67	67	337	525	590
SQL	time	0	1	2	5	6
	mem	65	126	154	289	289

and using different model query languages. This paper concludes with related work and conclusions in Sects. 5 and 6.

2 Operation with CDO Repositories

CDO provides transparent persistence of models in all kinds of back-end strategies, with load on demand mechanisms and caching policies to operate persisted models. CDO supports features such as: multi-user access, off-line collaboration, model-level locking, branching and versioning. Figure 1 illustrates the client/server architecture of CDO. **CDO Server** interacts with the database back-end through an IStore implementation. *DBStore* is the most mature and complete[2], and in practice mainly relational back-ends are used with CDO [3]. For relational back-ends, CDO provides a common data-schema with dedicated tables for change history, branches, commits or user access; additionally it generates automatically one data-schema for each different domain-metamodel.

EMF-based applications (editors, querying utilities, transformations, etc.) can operate with CDO repositories. For this purpose, **CDO Client** provides a custom extension of EMFs Resource (CDOResource) and EObjects (CDOObjects). Elements of the model are loaded in memory to operate them. CDO Client can also communicate with a server side query manager. CDO provides support for OCL queries (OCLQueryHandler) and SQL (SQLQueryHandler). Table 1 shows execution time and memory usage results for the GraBats case study [16] using Plain EMF, OCL, CDO-OCL and SQL on five CDO repositories with H2 relational database (Set0–4)[3]. Best results are obtained when operating from server-side query manager. In particular, CDOs' SQL query handler significantly improves performance of operation. However, the programming effort for model engineers to get complex queries right in SQL can be high, and they should be aware of database schema and persistence related issues.

[2] Comparison of CDO stores: http://goo.gl/cEemcL.
[3] For extended information about the evaluation, please refer to Sect. 4.

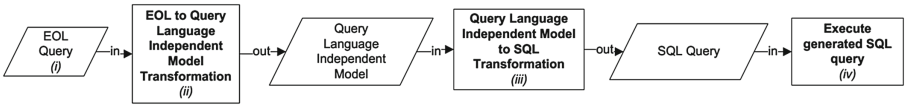


Fig. 2. EOL to SQL query transformation and execution process of CDO-QT.

3 CDO-QT

Some works have proposed query transformation from model query languages to SQL [5–7,10,13,14]. Inspired in these works, and with the aim of improving performance in CDO when operating with model query languages, we present Query Transformation Engine for CDO (CDO-QT).

CDO-QT inputs model traversal queries in a model query language (EOL [12]) and transforms automatically to SQL queries that are executed directly in CDO relational back-ends. Figure 2 illustrates the transformation process: (i) model engineers use EOL; (ii) CDO-QT transforms at runtime EOL statements into a language independent model; (iii) CDO-QT transforms the model into SQL statements; and (iv) SQL statements are executed directly over the database.

3.1 Query Language Independent Metamodel

CDO-QT uses a Query Language Independent Metamodel (QLI Metamodel) to specify queries in a language-independent way, separating the transformation to SQL from the model-query language. At this time, CDO-QT supports transformation of model traversal and self-contained EOL queries. A simplification of QLI Metamodel for model traversal EOL queries is illustrated in Fig. 3:

- `Query`: Root element of the model. Attribute `returnType` specifies the type of result returned by the query and `root` reference contains an `IModelTraversal` instance. `IModelTraversal` specifies statements that full-traverse models and is implemented by `AllInstancesOfKind` and `CollectionMethod`.
- `AllInstancesOfKind`. Abstraction for statements that traverse models searching instances of an specific kind (specified by `type`). Sample EOL statement is *MethodDeclaration.all*.
- `CollectionMethod`. Abstraction for query statements where all values of an input collection are evaluated (e.g. *.select(md:MethodDeclaration | ...)* EOL statement). `name` specifies type of the `CollectionMethod` (e.g. select, collect, etc.); `iterator` reference contains the input collection (`VariableIterator` instance); and `body` optional reference contains a `IQueryStatement` instance.
- `VariableIterator`. Specifies variables that iterate a collection values within a query. `type` contains `EClass` of the iterated values; `name` specifies the variable name; and `alias`, contains an unique name of the variable. `VariableIterator` contains an `ISource` instance (`source` reference). `ISource` is implemented by classes that specify collection of values iterated by a `VariableIterator` instance. Sample EOL statement is *md:MethodDeclaration*.

- `IQueryStatement` is an interface implemented by all classes that specify statements that could be contained by a `CollectionMethod` (`LogicalOP`, `ComparisonOP` and `Value`). `getType()` returns type of the value returned by the specified statement.
- `LogicalOP`. Abstraction for logical comparison of two statements[4] (contained by `left` and `right`) and returns a boolean value (`getType()=Boolean`). `Operator` specifies the logical operator type (AND, OR, NOT, etc.) Sample EOL statement is: `mod.private and mod.static`.
- `ComparisonOP`. Similar to `LogicalOP` but for comparison of values (e.g. EQUALS, LOWER, etc.). Sample EOL statement is `mod.private=true`.
- `Value`. Extended by `PrimitiveValue`, `CollectionMethod`, `ValueMethod` and `VariableValue` classes, and specifies statements returning a value.
- `PrimitiveValue`. Abstraction for primitive values (e.g. `true` boolean value).
- `ValueMethod`. For query statements that evaluate a single-value. `name` specifies method type; `params` contains parameters of the method; and `variable` reference contains a `VariableValue` instance which is evaluated. Sample EOL statement is md.isOfType(MethodDeclaration).
- `VariableValue`. Extended by `LocalVariableValue` and `VariableIterator` classes, is an abstraction for statements returning values derived from a variable specified within the query.
- `LocalVariableValue`. Abstraction for statements that specify value of a variable within the query. It contains `parentVariable` feature that references a `VariableValue` instance and `sf` attribute that specifies a `EStructuralFeature`. If `sf` is empty, this class specifies the value returned by the instance referenced at `parentVariable`. By contrast, if `sf` contains a feature, the class specifies the feature value in the `parentVariable` class values.

Figure 4 illustrates a sample QLI model that conforms to the QLI Metamodel. Section 3.3 describes this model and provides information about its generation.

3.2 CDO-QT Design

Figure 5 illustrates the class-diagram of CDO-QT:

- **Language independent** (`CDO-QT.generic` package). Classes and interfaces to be extended by EOL- and CDO-Specific packages. This design facilitates inclusion of new query languages. `MLDriver` transforms model queries into a language independent representation (`QLI Model`). `PLDriver` transforms into a database-specific language and executes the query.
- **EOL-Specific** (`CDO-QT.eol-specific` package). Deals with EOL and is responsible for parsing and transforming EOL queries into a QLI Model. `EOLDriver` extends `MLDriver` and implements `IModel` interface of the EMC API (provided by EOL) to interact with EOL queries. `generateQLIM()` method supports transformation of EOL queries into QLI model.

[4] With the exception of NOT operator that only has one statement contained by right reference.

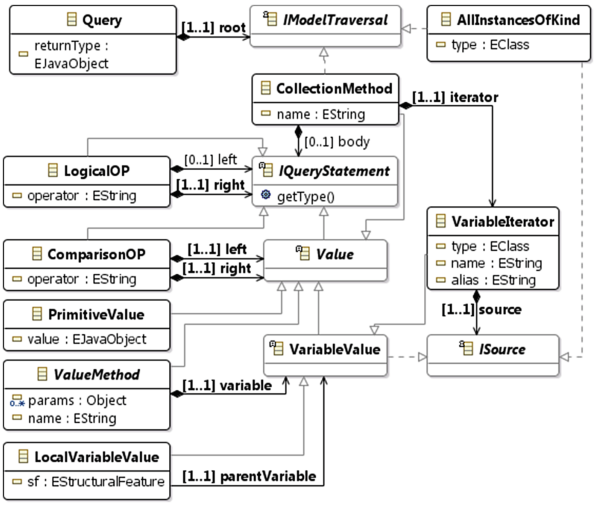


Fig. 3. Simplified QLI Metamodel

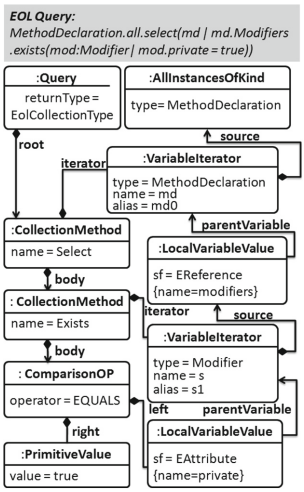


Fig. 4. QLI Model for a sample EOL query.

– **CDO-Specific** (`CDO-QT.dbstore-specific` package). Deals with CDO and is responsible for: (i) transforming QLI Model into a database specific language; and (ii) executing the query. We provide implementation for DBStore (SQL). `CDODriver` extends `PLDriver`. `generateQuery()` method that generates a SQL query from a language independent model. `execQuery()` method executes the generated SQL query through a `CDOQuery` instance (provided by CDO to execute SQL queries at server-side). CDODriver also implements `getVersionBranchInfo()` (which adds version and branch information to SQL queries) and `postProcessResults()` (for post-processing SQL results).

As shown in Fig. 6 user interacts with the `EOLDriver` (`execQuery` method). `EOLDriver` generates the `QLI Model` (`generateQLIModel` method) and calls `getResult` method of the `CDODriver`. `CDODriver` executes `generateQuery` method to obtain SQL query. Then, SQL query is completed with version/branch information (`addVersionBranchInfo()`). Next, query is executed, obtained results post-processed, and returned to `EOLDriver` and to the user.

3.3 From EOL to QLI Model

CDO-QT generates an intermediate and query language independent `QLI Model` from EOL queries: the `EOLDriver` receives from EOL an AST Tree that specifies the EOL Query, which is the input point of the transformation. Listing 1.1 illustrates a fragment of the transformation algorithm (`genQLIElem(AST n)`), where AST nodes are visited and artifacts of the `QLI Model` are instantiated. The algorithm is called recursively until all nodes are visited.

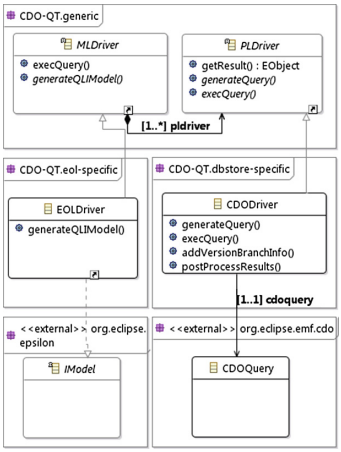


Fig. 5. CDO-QT class diagram

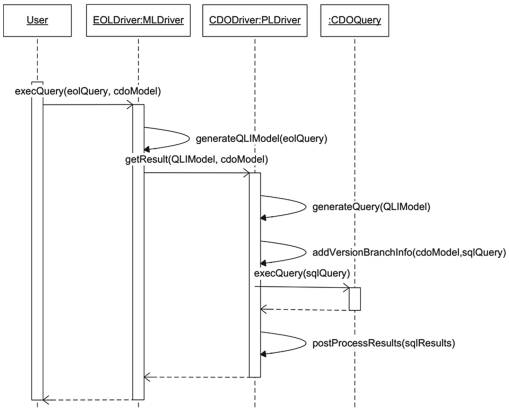


Fig. 6. CDO-QT sequence diagram

Following, QLI Model generation process for the EOL query illustrated on Fig. 4 is described: the transformation process starts with the AST node corresponding with the *MethodDeclaration.all* EOL statement. The transformation algorithm obtains the AST node and generates corresponding abstraction (`AllInstancesOfKind` instance). Next, AST node that specifies *.select(md — ...)* statement is processed by the algorithm and creates a `CollectionMethod` instance (with '`select`' name value). This instance contains a `VariableIterator` instance that specifies the collection iterator (*md*). `VariableIterator` iterates values returned by *MethodDeclaration.all* statement and consequently it contains the previously instantiated `AllInstancesOfKind`.

```
1 Object genQLIElem(AST n){
2 if ...
3 else if(n.hasChildren() && isComparison(n.getText()){
4   ComparisonOP obj = createComparisonOP();
5   obj.setLeft(genQLIElem(n.getFirstChild()));
6   obj.setRight(genQLIElem(n.getSecondChild()));
7   obj.setOperator(n.getText());
8   return obj;
9 } ... }
```

Listing 1.1. Fragment of the QLI Model generation algorithm.

The condition of the `CollectionMethod` instance is specified by *md.modifiers.exists(mod:Modifier | ...)* EOL statement and a `CollectionMethod` (named '`exists`') instance is created under the body reference. It contains a `VariableIterator` instance that contains a `LocalVariableValue` that specifies iterated values (*md.modifiers*). `CollectionMethod` body reference is filled with a `ComparisonOP` (abstraction of condition *mod.private=true* condition). The fragment shown in Listing 1.1 contains code related to the generation of

the `ComparisonOP` instances. In this case, it satisfies `n.hasChildren()` condition of line 3 (one child for each compared side) and `n.getText()` of line 3 returns the string `=` satisfying also second condition. Satisfying conditions involves the instantiation of a new `ComparisonOP`(line 4). Left and right references are obtained executing the algorithm `genQLIElem(AST n)` for two children (lines 5,6). In this case, `left` contains a `LocalVariableValue` that specifies *mod.private* statement, and `right` contains a `PrimitiveValue` instance that specifies *true* boolean value. Finally, operator feature is setted with the value returned by `n.getText()`(line 7) and the instantiated element is returned (line 8).

3.4 From QLI Model to SQL

`EOLDriver` calls `CDODriver` passing by arguments the `QLI Model` and a `CDO-Resource` instance (queried model). At this point, the prototype uses the default mapping strategy of the DBStore (horizontal mapping). We can distinguish two different types of tables within the domain-specific data-schema: (a) **Object-Tables**: contain information about all the instances of an specific type. The name of the each table corresponds with name of the type of the containing elements (e.g. *MethodDeclaration*); (b) **Many-Value-Ref-Tables**: contain information about a many-value reference of an specific type. The name of the table will follow this format: TypeName_FeatureName_List (e.g. *MethodDeclaration_BodyDeclarations_List*).

Table 2 describes a simplified version of SQL queries that are generated from each `QLI Model` element. Branching and versioning related statements are added in generated SQL queries:

- ***WHERE* statements that obtain information from an Object-Table.** Following, simplified version of the added SQL statement that is described: `CDO_VERSION>0 AND ((CDO_BRANCH =:branchID AND CDO_CREATED <= :commit AND (CDO_REVISED=0 OR CDO_REVISED>:commit)) OR (:hasBase AND CDO_BRANCH =:baseID AND CDO_CREATED<=:basetime AND (CDO_ REVISED=0 OR CDO_REVISED>:basetime)))`. The statement contains the following parameters: (1) `commit`, specifies the timestamp of the commit corresponding with the model version; (2) `branchID`, specifies the identifier of the branch that is being queried; (3) `hasBase`, boolean value that specifies if the branch is based in another branch; (4) `baseID`, specifies the identifier of the base branch; and (5) `baseTime`, specifies the timestamp of the corresponding version of the base branch.
- **INNER JOIN statements that join an Object-Table with a Many-Value-Ref-Table.** This is a simplified version of the SQL statement that is added: `objectTable.CDO_VERSION = referenceTable.CDO_VERSION AND objectTable.CDO_BRANCH = referenceTable.CDO_BRANCH`.

Table 2. SQL queries generated for each QLI model element.

QLI Element	Generated SQL
AllKindInstances	types: `SELECT * FROM TypeTable WHERE ...` subtypes: `(SELECT * FROM SubType1Table)` `UNION (SELECT * FROM SubType2Table) ...`
LogicalOP	`(rightStatementSQL (AND | OR | ...) leftStatementSQL)`
ComparisonOP	`(rightStatementSQL ( = | < | ...)leftStatementSQL)`
PrimitiveValue	strings: *'value'*; other types: *value*

ValueMethod	method-specific SQL. Ex.: var.feature.isTypeOf(type):
	`EXISTS(SELECT * FROM TypeTable AS T`
	`WHERE var.feature=T.CDO_ID AND ...)`

CollectionMethod	method-specific SQL. Ex.: var.feature.exists(it — cond):
	`EXISTS (SELECT iteratorName.*`
	`FROM (VariableIteratorSQL) AS iteratorName`
	`WHERE condSQL)`

VariableIterator	`IteratorParentType_IteratorParentFeature_List` `INNER JOIN (iteratorType) AS iteratorName ON`

LocalVariableValue	multi-value refs: `SELECT featureName.CDO_VALUE`
	`FROM ParentType_Feature_List`
	`INNER JOIN FeatureType AS featureName ON`
	`WHERE ParentType_Feature_List.CDO_SOURCE =`
	`parent.CDO_ID AND ... )`
	attributes and single-value refs: `SELECT parent.feature`
	`FROM ParentTable WHERE l`

3.5 Executing the Query

Version and branch parameters are set using the `CDOResource`. Listing 1.2 illustrates the parameter setting process: (1) parameter values are obtained from the `CDOResource` instance; (2) obtained values are set to the generated SQL through the `CDOQuery` instance (`cqo`); and (3) the SQL query is executed over the CDO repository using the `CDOQuery` class provided by CDO. Obtained results correspond to all the models (`CDOResource`) of the repository. To provide results for a specific model, CDO-QT filters and/or analyses the SQL results. For example, to obtain all *MethodDeclaration* instances, the post-process selects those that are part of the model (`object.cdoResource() == resource`). To check if a `MethodDeclaration` exists in a model, SQL results are analysed (e.g. `while(res.hasNext()){ if (res.getNext().cdoResource == resource) return true;} return false;`). We have decided to do this post-process as including it in the transformation would require complex SQL queries that could have impact in performance.

```
void setQueryParameters(CDOResource resource, CDOQuery cqo){
 boolean hasParent = false;
 long commit = getTimeStamp(resource.cdoView)
 long branchID = resource.cdoView().getBranch().getID();
 if(existsBase(resource)) hasParent = true;
 long baseID = getBaseID(resource);
 long baseTime = getBaseTime(resource);
 cqo.setParameter("commit", Long.toString(commit));
 cqo.setParameter("branchID", Long.toString(branchID));
 cqo.setParameter("hasParent", hasParent);
 cqo.setParameter("baseID",Long.toString(baseID));
 cqo.setParameter("baseTime",Long.toString(baseTime));}
```

Listing 1.2. Setting paramater values of the generated SQL queries.

4 Evaluation

All the experiments have been executed as a standalone application over a Microsoft Azure[5] virtual machine configured with a 4 Core processor, 14 GB of RAM, 200 GB SSD, and running 64-Bit Windows Server 2012 and Java SE v1.8.0. We have used Eclipse Mars with CDO 4.4. CDO repositories have been executed in embedded mode[6] to measure total memory usage and avoid the uncertainty of connections in the execution time. Repositories run on top of H2 v1.3.168, using the DBStore with its default mapping, caching and pre-fetching values, and supporting audits and branches.

Correctness of query-results has been ensured by automatically comparing the results of each query using different languages. In order to get reliable numbers, each query was processed 5 times for each evaluation case and Java Virtual Machine has been restarted for each execution. Results have been evaluated against the following quantitative metrics: M1: Average Execution Time (in seconds) and M2: Maximum Memory Usage (in MB). M2 includes memory used by the CDO Client and Server. We have used three different queries in the evaluation: Q1: Number of classes (TypeDeclaration instances) existing within the model, Q2: Number of private methods (MethodDeclaration instances) existing within the model and Q3: Number of singletons (TypeDeclaration instances) existing within the model (GraBaTs case study query). All queries traverse model but with increasing complexity. We have expressed queries in Plain EMF, OCL (used by MDT OCL and CDO-OCL), SQL and EOL (used by CDO-QT). We have used metamodel and model instances from the GraBaTs 2009 case study [16]. Models specify source code of different Java packages and conform to the *JDTAST* metamodel which contains abstractions of the Java source code. Table 3 shows results of queries for each model.

In this evaluation we address: *Which is the performance of querying models within a CDO repository using EMF Plain, MDT OCL, CDO_OCL, SQL*

[5] Azure: https://azure.microsoft.com/en-us/services/virtual-machines/.
[6] CDO/Embedded: https://wiki.eclipse.org/CDO/Embedded.

Table 3. Properties of the GraBaTs models.

	XMI size	Repository size	Numb. of models	Model Elem	Q1 Results	Q2 Results	Q3 Results
Set0	8,8	15.3 MB	1	70447	14	4	1
Set1	27	43.8 MB	1	198466	40	38	2
Set2	271	307 MB	1	2082841	1605	1793	41
Set3	598	784 MB	1	4852855	5314	9275	155
Set4	646	1.17 GB	1	4961779	5984	10086	164
Set5	n/a	2.01 GB	2	9923558	5984	10086	164
Set6	n/a	2.88 GB	3	14885337	5984	10086	164
Set7	n/a	3.67 GB	4	19847116	5984	10086	164
Set8	n/a	4.45 GB	5	24808895	5984	10086	164
Set9	n/a	5 GB	6	29770674	5984	10086	164

and CDO-QT? We distinguish two different configuration factors (F) that may impact:

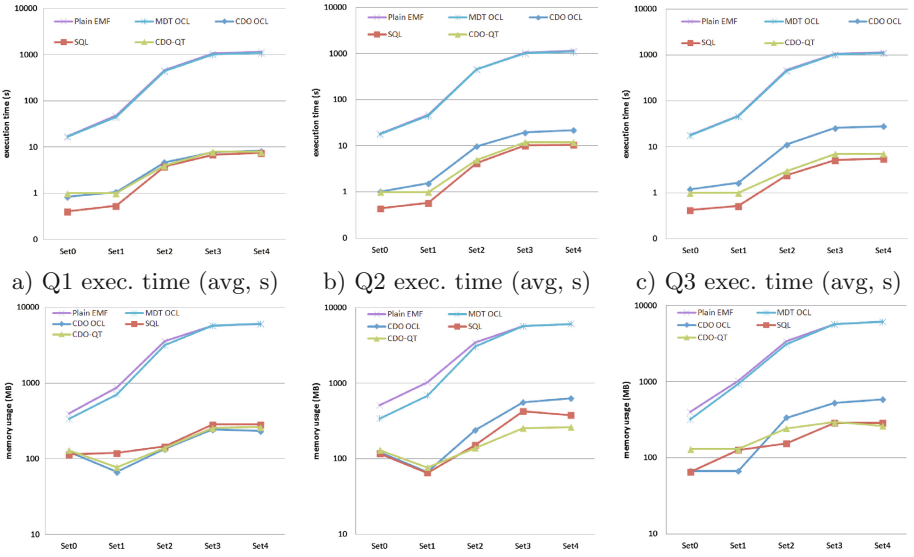
- **F1, Size of the model**: We measure how the increasing size of the model may influence on the performance (execution time and memory). We measure the size of a model in number of elements. For this factor, each model has been persisted in a different CDO Repository (from Set0 to Set4 of Table 3).
- **F2, Size of the repository**: As in CDO we can save many models in the repository, we have measured how the increasing size of the repository may influence on the performance. We measure the size of the repository in number of models and elements within the repository. For this factor, we have stored set4 model copies within the same CDO Repository (from Set4 to Set9 of Table 3).

Extended information in http://xdecarlos.bitbucket.org/fase_2016/.

4.1 Discussion

F1: Model-Size Influence (Set0-Set4). Size of the queried model has a great impact over the time and memory required in Plain EMF and MDT OCL, and three queries result in similar values (entire model is always loaded in memory). In Set4, these client-side solutions require more than 6000 % of time of Set0 and more than 1100 % of memory. *Plain EMF* requires 17-18 s and 396-513 MB for querying the smallest model (Set0) and 1140-1166 s and 6-6.1 GB for the largest (Set4). Model size impact is slightly lower for *MDT OCL* as it requires 17-18 s and 322-342 MB for Set0 and 1090-1101 s and 6-6.1 GB for Set4.

Figure 7 illustrates time and memory results and they show that, the impact of the model size is lower if queries are executed at server-side. In Set4, these



a) Q1 exec. time (avg, s) b) Q2 exec. time (avg, s) c) Q3 exec. time (avg, s)

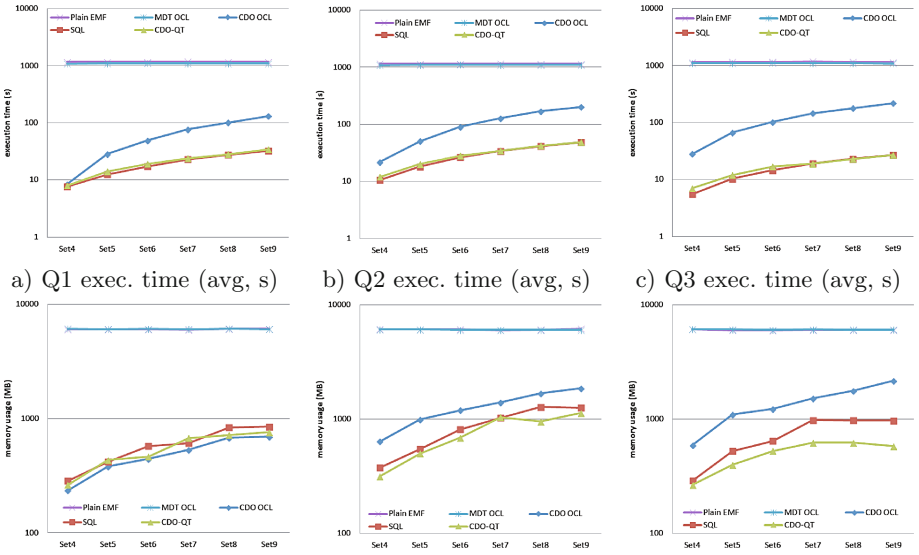
d) Q1 memory (max, MB) e) Q2 memory (max, MB) f) Q3 memory (max, MB)

Fig. 7. Execution time and memory results of queries from Set0 to Set4.

solutions require up to 2400 % of time of Set0 and up to 800 % of memory. However, increase values are much lower than on client-side solutions.

CDO-OCL is more than 17 times faster than Plain EMF and MDT OCL, and it only requires 1 s for executing queries in Set0. Memory usage is also reduced to 123 MB (Q1-Q2) and 67 MB (Q3). In case of other sets, results vary depending on the query: Q1 requires less time than Q2 and Q3, and Q2 less than Q3. For example, in Set4 Q1 requires 8 s, Q2 22 s and Q3 28 s. However, Q3 is more than 38 times faster than any query in Plain EMF or MDT OCL. In terms of memory, Q1 requires less than Q2 and Q3: in Set4 Q1 needs 235 MB, Q2 636 MB and Q3 590 MB. Worst memory value (636 MB) is more than 9 times lower than the best memory usage result of the client-side solutions. *SQL* shows better results: queries require less than a second and 118 MB in Set0; and less than 12 s and 375 MB in Set4. Q1 requires less time and memory than Q2 and results are similar of CDO-OCL. In the case of Q2 it is 2 times faster than CDO-OCL and memory usage is reduced by 40 % for Set4. Q3 time and memory results are lower than Q1 and Q2, and it is more than 4 times faster than CDO-OCL requiring less than 50 % of memory.

Performance and memory results of *CDO-QT* for executing queries using EOL are similar to SQL. Execution time results show that CDO-QT requires between 1 and 2 s more than SQL to be executed. The generated SQL query is the same that is used in the SQL experiments, and it indicates that the extra-time corresponds with the EOL to SQL transformation. CDO-QT requires 1 s and less than 130 MB for executing queries in Set0, and less than 8 s and 315 MB



a) Q1 exec. time (avg, s) b) Q2 exec. time (avg, s) c) Q3 exec. time (avg, s)

d) Q1 memory (max,MB) e) Q2 memory (max,MB) f) Q3 memory (max,MB)

Fig. 8. Execution time and memory results of queries from Set4 to Set9.

in Set4. As occurs in SQL, Q3 requires less time and memory than Q1 and Q2, and Q1 less than Q2. For example in Set4: Q1 requires 8 s and 263 MB, Q2 12 s and 315 MB, and Q3 7 s and 263 MB. CDO-QT results are significantly better than using the other server-side solution (CDO-OCL), and much better than using a client-side solution (Plain EMF and MDT-OCL).

F2: Repository-Size Influence (Set4-Set9). Time and memory results obtained querying F2 models (set4-set9) indicate that the size of the repository has not influence in queries executed at the client-side: In the case of *Plain EMF* execution time value for executing queries is between 1140-1174 s and requires around 6 GB of memory; in the case of *MDT OCL* the execution time is slightly lower (between 1081-1113 s) and also requires around 6 GB of memory.

As Fig. 8 illustrates, this scenario changes in case of the server-side solutions, where the size of the repository has influence. *CDO-OCL* results show a constant increase of the query execution time from one repository to the subsequent one (e.g. from Set5 to Set6). The increase changes according to query: between 20-28 s for Q1, 31-41 s for Q2, and 35-42 s for Q3. Memory usage increases: from 235 MB to 695 MB in Q1; from 636 MB to 1860 MB in Q2; and from 590 MB to 2171 MB of Q3. The influence of the repository size is greater in Q3, which requires more time and memory. In Set4, CDO-OCL requires around 1100 % of time of Set0 and around 330 % of memory. The trend is similar in *SQL*, but the time increase between repositories is lower: 4-6 s for Q1, 7-8 s for Q2, and 4-5 s for Q3. Memory values increase from 285 MB to 849 MB for Q1; from 375 MB

to 1249 MB for Q2; and from 289 MB to 968 MB for Q3. In the case of SQL, repository-size influence is greater in Q2. Q3 is resolved faster and Q1 requires less memory than others. In Set4, SQL requires around 430 % of time of Set0 and around 300 % of memory. While increase is similar to CDO-OCL in case of memory, increment of the execution time is lower.

CDO-QT results agree with those obtained in SQL and execution time and memory is also influenced by repository size, but it is lower than in CDO-OCL. Execution time difference between SQL and CDO-QT is only of 1-2 s (transformation time overhead). In terms of memory, CDO-QT uses less memory than others (including SQL): from 263 MB to 761 MB for Q1, from 315 MB to 1136 MB for Q2, and from 264 MB to 579 MB for Q3. The filtering mechanism provided by CDO-QT could be the reason of memory usage difference between SQL and CDO-QT. Results show that the execution time and memory usage of CDO-QT is much lower than the required by the client-side solutions (Plain EMF and MDT OCL). Additionally, CDO-QT resolves these queries faster than the natively provided server-side version of OCL (CDO-OCL).

4.2 Threats to Validity

All the queries full traverse the model, therefore they start the computation by obtaining all the instances of an specific type that exists within the queried model. This type of queries covers the majority of computational-demanding queries in real industrial domains such as reverse engineering domain [9]. However, there are other types of queries (e.g. non-traversal queries or queries that modify the model) that have not been tested. Moreover, models have been generated for test-case purpose. Using industrial models and real model operations would be more realistic. We plan to perform it in a future version of this work.

5 Related Work

Query Transformation. [10] describes a framework that supports mapping of UML models to arbitrary data-schemas and mapping of OCL invariants to a declarative query language. [14] transforms OCL constraints into SQL to check integrity of UML models persisted in relational repositories. [7] generates SQL queries from OCL constraints and executes over MySQL databases. [6] provides a tool based on OCL2SQL that generates views from OCL constraints. All these approaches provide generation at compilation-time from OCL (declarative language) to SQL (declarative language). By contrast, CDO-QT transforms at run-time EOL, an imperative model-level language, into a declarative persistence-specific query language (SQL).

CDO Evaluation. [15] includes evaluation of CDO by comparing results of performing different model operations (store, query and modify) with XMI and Morsa. [2] describes the performance and memory usage required by different persistence mechanisms (Teneo, CDO, Neo4J and OrientDB) for executing the GraBaTs case study query. These studies show CDO results for three models of

GraBaTs (Set0–2), by contrast, our evaluation includes results of all the GraBaTs models (Set0–4). [3,9,15] include an analysis of CDO and other persistence mechanisms through the execution of different types of queries. While they use one query language for executing queries in CDO, our study shows results for different query languages (Plain EMF, MDT OCL, CDO OCL, SQL and EOL with CDO-QT). [8] focuses the evaluation in the model query languages and describes the GraBaTs query results using different query languages and persistences (XMI, CDO and MORSA). While the GraBats query is included in this study, we have executed two additional queries using different query languages that are executed against CDO repositories.

Improve Query Performance. EMF-IncQuery provides support for executing model queries in an incremental way only over model parts that have changed [18]. [19] focuses on improving efficiency of model traversal EOL queries. While these approaches provide improvements on user-side query execution, CDO-QT provides support for generating SQL queries that are executed over persistence (relational back-end) and at server-side.

6 Conclusions and Future Work

In this paper, we have presented CDO-QT, an approach that: (i) provides a two-step transformation process that generates SQL queries from EOL queries; and (ii) executes generated SQL at server-side over CDO repositories. CDO-QT is able to execute model traversal queries in a model query language (EOL), but with a performance similar to SQL. We have compared the performance and memory usage results of executing different model query languages: Plain EMF, MDT OCL, CDO OCL, SQL and EOL using CDO-QT. GraBaTs 2009 Case Study models have been persisted in different CDO repositories with size from 15.3 MB to 5 GB. Execution time and memory results show that CDO-QT is a promising alternative for making queries from EOL to CDO repositories. Results indicate that CDO-QT is much faster and use less memory than model query languages executed at client-side of CDO (Plain EMF and MDT OCL). Moreover, obtained results are better than the natively supported CDO-OCL that executes OCL queries at server-side.

This prototype of CDO-QT provides support for executing self-contained and model traversal EOL queries. However, we plan to extend it to support more types of EOL queries (e.g. non-traversal queries, queries that modify models, query chains, etc.). For future work, we plan to provide CDO-QT implementations of additional model query languages, supporting transformation of other types of languages (e.g. IncQuery or OCL). We also plan to provide implementations for other stores of CDO and for other persistence mechanisms.

Acknowledgements. The authors wish to thank Xabier Mendialdua for his contributions. 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. Bagnato, A., Brosse, E., Sadovykh, A., Maló, P., Trujillo, S., Mendialdua, X., De Carlos, X.: Flexible and scalable modelling in the MONDO project: industrial case studies. In: Proceedings of the 3rd Workshop on Extreme Modeling Co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems, XM@MoDELS 2014, 29 September 2014, Valencia, Spain, pp. 42–51 (2014)
2. Barmpis, K., Kolovos, D.S.: Evaluation of contemporary graph databases for efficient persistence of large-scale models. J. Object Technol. **13**(3), 3:1–3:26 (2014)
3. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a scalable persistence layer for EMF models. In: Cabot, J., Rubin, J. (eds.) ECMFA 2014. LNCS, vol. 8569, pp. 230–241. Springer, Heidelberg (2014)
4. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010)
5. De Carlos, X., Sagardui, G., Murguzur, A., Trujillo, S., Mendialdua, X.: Model query translator - a model-level query approach for large-scale models. In: MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, 9–11 February 2015, Angers, Loire Valley, France, pp. 62–73 (2015)
6. 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. LNCS, vol. 2185, pp. 104–117. Springer, Heidelberg (2001)
7. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: a stored procedure-based MySQL code generator for OCL. Electron. Commun. EASST **36**, 1–16 (2010)
8. Pagán, J.E., Molina, J.G.: Querying large models efficiently. Inf. Softw. Technol. **56**(6), 586–622 (2014)
9. Gómez, A., Tisi, M., Sunyé, G., Cabot, J.: Map-based transparent persistence for very large models. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 19–34. Springer, Heidelberg (2015)
10. Heidenreich, F., Wende, C., Demuth, B.: A framework for generating query language code from OCL invariants. Electron. Commun. EASST **9**, 1–10 (2007)
11. Kärnä, J., Tolvanen, J.-P., Kelly, S.: Evaluating the use of domain-specific modeling in practice. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (2009)
12. Kolovos, D.S., Rose, L., Garcia-Dominguez, A., Paige, R.: The Epsilon Book. Eclipse, Newyork (2010)
13. Kolovos, D.S., Wei, R., Barmpis, K.: An approach for efficient querying of large relational datasets with OCL based languages. In: Proceedings of the Workshop on Extreme Modeling co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2013), 29 September 2013, Miami, Florida, USA, pp. 46–54 (2013)
14. Marder, U., Ritter, N., Steiert, H.P.: A DBMS-based approach for automatic checking of OCL constraints. Proc. OOPSLA **99**, 1–5 (1999)
15. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A repository for scalable model management. Softw. Syst. Model. **14**, 1–21 (2013)
16. Sottet, J.-S., Jouault, F., et al.: Program comprehension. In: Proceedings of the 5th International Workshop on Graph-Based Tools (2009)

17. Stepper, E.: CDO (2009). http://eclipse.org/cdo/. Accessed 30 January 2015
18. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries. Sci. Comput. Program. **98**, 80–99 (2015)
19. Wei, R., Kolovos, D.S.: An efficient computation strategy for allInstances(). In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering Part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, 23 July 2015, L'Aquila, Italy, pp. 32–41 (2015)