Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

An Approach for Efficient Querying of Large Relational Datasets with OCL-based Languages

Ran Wei[a] Konstantinos Barmpis[b] Dimitrios Kolovos[b]

a. Dalian University of Technology, China
http://www.dlut.edu.cn

b. University of York, United Kingdom
http://www.york.ac.uk

Abstract Model Driven Engineering (MDE) is increasingly applied to engineering processes due to benefits of automation introduced by MDE. Relational database management systems are used to store and manage large sets of data, subsets of which can be of interest in MDE engineering processes. To enable seamless integration of information stored in relational databases in an MDE process, the technical and conceptual gap between the two technical spaces needs to be bridged. In this paper we investigate the challenges involved in querying large relational datasets using an imperative OCL-based model management language (EOL) through running examples, and we propose solutions for some of these challenges.

1 Introduction

Information that can potentially be of interest in the context of a Model Driven Engineering (MDE) process is often located within non-model artefacts such as spreadsheets, XML documents and relational databases. As such, model management languages and tools would arguably benefit from extending their scope beyond the narrow boundaries of 3-level metamodelling architectures such as EMF and MOF for MDE.

In previous work, we have demonstrated how OCL-based model management (e.g. model validation, model-to-text and model-to-model transformation) languages of the Epsilon platform [Ric09] can be used to interact with plain XML documents [Dim12] and spreadsheets [Mar13]. In this work we investigate the challenges involved in using such languages to query large relational datasets and extract abstract models that can be then used (e.g. analysed, validated, transformed) in the context of MDE processes. In particular, we identify the challenges imposed by the size of such datasets

Ran Wei, Konstantinos Barmpis, Dimitrios Kolovos. *An Approach for Efficient Querying of Large Relational Datasets with OCL-based Languages.* Licensed under . In Journal of Object Technology, vol. 0, 2010, pages 0:1–9. Available at http://www.jot.fm/

and the conceptual gap between the organisation of relational databases and the object-oriented syntax of OCL-based languages, and we propose some solutions.

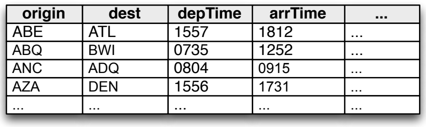
The rest of the paper is organised as follows. In Section 2 we present a running example that involves querying a real-world large relational dataset and extracting an EMF model from it using an OCL-based imperative transformation language, we identify the performance challenges involved in doing so, and propose a run-time query translation approach that addresses some of these challenges. In Section 3, we review previous work on using OCL to query relational datasets and compare our approach to it, and in Section 4 we conclude the paper and provide directions for further work.

2 Querying Large Relational Datasets: Challenges and Solutions

The Epsilon Object Language [Dim06] is an OCL-based imperative model query and transformation language. EOL is the core language of the Epsilon platform and underpins a number of task-specific languages for model management tasks including model validation, model-to-model and model-to-text transformation. As such, by adding support for querying relational datasets to EOL, this capability is automatically propagated to all task-specific languages of the platform. While the discussion in the rest of the paper focuses on EOL, in principle the discussion and solutions proposed are also relevant to a wide range of OCL-based model management languages such as QVTo, ATL and Kermeta.

To experiment with querying relational datasets with EOL, we selected a large real-world publicly-available dataset from the US Bureau of Transportation Statistics[1] that records all domestic flights in the US in January 2013. The dataset consists of one table (Flight) with 223 columns and 506,312 rows and is 221MB when persisted in MySQL. Each row of the table records the details of a domestic flight during that month, including the short codes of its origin and destination airports, the flight's departure and arrival time etc. An excerpt of the Flight table appears in Figure 1.

Our aim in this running example is to transform this dataset into an EMF model that conforms to the metamodel of Figure 2 and which captures the incoming and outgoing routes for each airport as well as the volume of traffic on these routes, so that we can then further process the EMF model to discover interesting facts about the structure of the US airport network.



origin	dest	depTime	arrTime	...
ABE	ATL	1557	1812	...
ABQ	BWI	0735	1252	...
ANC	ADQ	0804	0915	...
AZA	DEN	1556	1731	...
...	...	...	...	...

Figure 1 – Excerpt of the Flight table

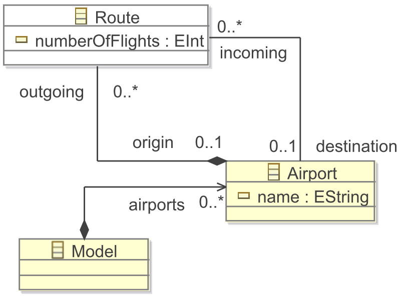


Figure 2 – Simple ATM System Metamodel

[1] http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

2.1 Finding the number of airports in the network

A reasonable OCL-like expression[2] that can be used to retrieve the number of all distinct airports[3] in the Flight table would be:

```
ht.allInstances.origin.asSet().size()
```

When such an expression is evaluated against an in-memory model (e.g. an EMF model) the EOL execution engine performs the following steps:

1. It inspects the model and computes a collection of all model elements of type Flight;

2. It iterates through the contents of the collection computed in step 1 and collects the values of their *origin* properties in a new collection;

3. It removes all duplicates from the collection computed in step 2;

4. It computes the size of the collection computed in step 3.

To evaluate the same expression against the relational database discussed above, we can assume that each table in the database is a type and each row in the table is a model element that is an instance of that type. Under these assumptions, the following issues emerge:

1. To compute the *Flight.allInstances* collection, the engine needs to execute the following SQL query: *select * from Flight.* Due to the size of the Flight table, the returned result-set cannot fit in a reasonable amount of heap space (we experimented with up to 1GB), and as such it needs to be streamed from the database to the engine instead. Streamed result-sets demonstrate the following challenges:

   They support forward-only iteration;

   To calculate the size of a streamed result-set it needs to be exhaustively iterated (in which case it becomes unusable as only forward iteration is permitted);

   Each database connection cannot support more than one streamed result-sets at a time.

2. Iterating through all rows of the Flight table through a streamed result set and collecting the values of the *origin* column of each row is particularly inefficient given that the same result can be achieved at a fraction of the time using the following SQL statement: *select origin from Flight*;

3. Eliminating duplicates by iterating the collection computed in step 2 is also inefficient as the same result can be achieved using the following – more efficient – SQL statement: *select distinct origin from Flight*;

4. Finally, calculating the size of a streamed result-set is not trivial without invalidating the result-set itself. By contrast, this could be computed in one step using the following SQL statement: *select count(distinct origin) from Flight.*

[2] EOL does away with the ocl- prefixes (e.g. oclAsSet()) and the → OCL operator and uses . instead for all property/method calls.

[3] We assume that there are no airports with only incoming or outgoing flights and as such, looking into one of *origin*, *dest* should suffice.

2.2 Finding adjunct airports

Assuming that we have computed a set containing the short codes of all airports in the table, the next task is to find for each airport, which other airports are directly connected to it, and then compute the volume of traffic between each pair of adjunct airports. An imperative EOL program that can be used to achieve this follows:

```
1 origins = Flight.allInstances.origin.asSet();
2 (origin in origins) {
3     var destinations = Flight.allInstances.dest.asSet();
4     for (destination in destinations) {
5           var numberOfFlights = Flight.allInstances.
6                 select(f|f.origin = origin and f.dest = destination).
7                       size();
8     }
```

The following observations can be made for the program above:

- Although the *destinations* result-set computed in line 3 does not change, it needs to be re-computed for every nested iteration as the result of the computation is streamed, and therefore only permits forward navigation;
- Unless care is taken to evaluate the right-hand side expressions in lines 1 and 3 using different database connections, the program will fail (as discussed above, each MySQL connection only permits at most one streamed result-set at a time);
- Iterating through all the rows of the Flight table in the *select(. . .)* method in lines 5-7 is inefficient, particularly as the same result can be computed using the following SQL statement *select count(*) from Flight where origin=? and destination=?* (where *?* should be replaced every time with the appropriate origin/destination values).

2.3 Runtime SQL Query Generation

In this section we argue that while the naive way of evaluating OCL-like queries on relational datasets can dramatically degrade performance (as shown in the previous section), there are certain runtime optimisations that the execution engine can perform to significantly reduce the execution time and memory footprint of some types of queries.

After applying such optimisations, the following EOL transformation, can transform the complete dataset (DB) in question to an EMF-based model (ATMS) that conforms to the metamodel of Figure 2 in less than 45 seconds on average hardware[4]. A visualisation of an excerpt of the extracted model appears in Figure 3.

The functionality of the transformation is outlined below:

- In line 1 it creates a new instance of the *Model* EClass in the ATMS EMF (target) model;
- In line 2 it computes a set of all origin airports in the Flight table;
- In line 4 it iterates through the set of strings computed in line 2;

[4] CPU: 2.66 GHz Intel Core 2 Duo, RAM: 8 GB DDR3.

- In line 5 it invokes the *airportForName* method defined in lines 21-30 which returns an instance of the *Airport* EClass in the target model with a matching name;
- In lines 6-7 it computes a set of adjunct airports to the origin airport;
- In lines 10-12 for each adjunct airport (destination), it computes the number of flights between the two airports;
- In lines 13-16 it creates a new instance of the *Route* EClass in the target model and populates its origin, destination and numberOfFlights properties.
- The *airportForName()* method in lines 21-30 is responsible for preventing the creation of airports with duplicate names in the target model.

```
 1  m : new ATMS!Model;
 2  origins = DB!Flight.allInstances.origin.asSet();
 3
 4  (origin in origins) {
 5      var originAirport = airportForName(origin);
 6      var destinations = DB!Flight.allInstances.
 7             select(f|f.origin = origin).dest.asSet();
 8
 9      for (destination in destinations) {
10             var numberOfFlights = DB!Flight.allInstances.
11                    select(f|f.origin = origin and f.dest = destination)
12                           .size();
13             var route = new ATMS!Route;
14             route.origin = originAirport;
15             route.destination = airportForName(destination);
16             route.numberOfFlights = numberOfFlights.asInteger();
17      }
18
19
20
21  ation airportForName(name : String) {
22      var airport = ATMS!Airport.allInstances.
23             selectOne(a|a.name = name);
24
25      if (airport.isUndefined()) {
26             airport = new ATMS!Airport;
27             airport.name = name;
28             m.airports.add(airport);
29      }
30      return airport;
```

Listing 1 – EOL transformation

To achieve an acceptable level of performance, we have extended the EOL execution engine to use streamed *lazy collections* and a runtime OCL to SQL query translation strategy for certain types of OCL expressions when the latter are evaluated against relational datasets. Each lazy collection acts as a wrapper for an SQL query generated

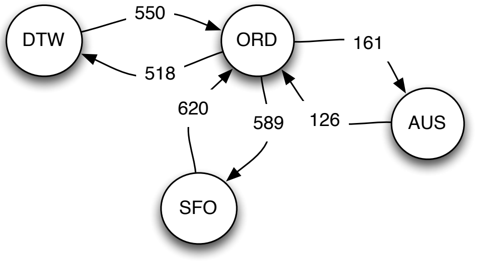


Figure 3 – Visualisation of an excerpt of the model extracted using the transformation in Listing 1

at runtime and only starts streaming data from the database if/when it needs to be iterated. This prevents unnecessary database queries and enables multi-step query translation at runtime. An example of the query translation process is illustrated in Figure 4 which calculates the average delay of flights flying from JFK to LAX on Sundays. In particular, the following OCL expressions are rewritten as SQL queries.

.allInstances Retrieving all the rows of a table in the database returns a streamed lazy collection (*ResultSetList*) that is backed by a *select * from <table>* SQL expression. For example, *Flight.allInstances* is translated to *select * from Flight.*

.select(<iterator>|<condition>) *ResultSetList* overrides the built-in OCL select operation, translates the EOL condition to an SQL expression, and returns a new *ResultSetList* constrained by the latter. For example, *Flight. allInstances.select(f |f.origin = "JFK" and f.dayOfWeek=1)* is translated into *select * from Flight where origin = ? and dayOfWeek = ?* (the values of the parameters – i.e. JFK and 1 – are kept separately and are only used if the query needs to be executed). The condition can contain references to the columns of the table, arithmetic, and logical operators at arbitrary levels of nesting. The *exists()*, *forAll()* and *reject()* OCL operations behave similarly.

.collect(<iterator>|<expression>) *ResultSetList* overrides the built-in OCL collect() operation to return a streamed lazy collection of primitive values (*PrimitiveValuesList*). For example, *Flight.all.collect(f |f.dest + "-" + f.origin)* is translated to *select origin + "-" + dest from Flight.* In the spirit of OCL, retrieving properties of collections is a short-hand notation for collect(). For example, *Flight.allInstances.dest* is shorthand for *Flight.allInstances. collect(f |f.dest).*

.size() Calls to the size() method of a *ResultSetList/PrimitiveValuesList* are interpreted as *count* SQL queries. For example *Flight.allInstances.size()* is translated to *select count(*) from Flight.*

asSet() Calls to asSet() method of a *PrimitiveValuesList* return a new *PrimitiveValuesList* backed by a *distinct* SQL query. For example, *Flight.allInstances. dest.asSet()* returns a new *PrimitiveValuesList* backed by the following SQL query: *select distinct(dest) from Flight.*

Streamed lazy collections also provide a *fetch()* method that executes their underpinning query and returns a complete in-memory result-set which is navigable in both

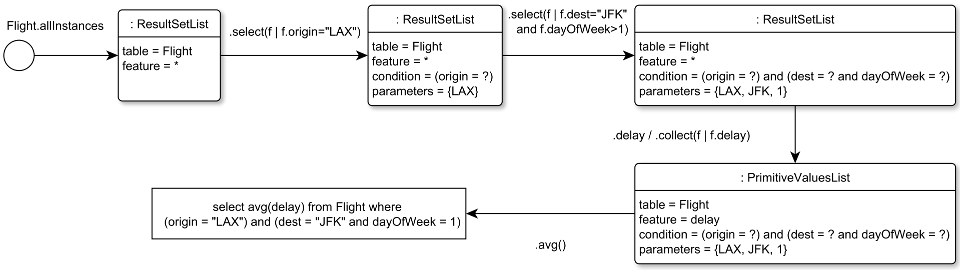

Flight.allInstances
: ResultSetList
table = Flight
feature = *
.select(f | f.origin="LAX")
: ResultSetList
table = Flight
feature = *
condition = (origin = ?)
parameters = {LAX}
.select(f | f.dest="JFK"
and f.dayOfWeek>1)
: ResultSetList
table = Flight
feature = *
condition = (origin = ?) and (dest = ? and dayOfWeek = ?)
parameters = {LAX, JFK, 1}
.delay / .collect(f | f.delay)
: PrimitiveValuesList
table = Flight
feature = delay
condition = (origin = ?) and (dest = ? and dayOfWeek = ?)
parameters = {LAX, JFK, 1}
.avg()
select avg(delay) from Flight where
(origin = "LAX") and (dest = "JFK" and dayOfWeek = 1)

Figure 4 – Multi-step SQL query generation process

directions. This is useful for small result-sets where the overhead of maintaining the entire result-set is memory is preferable to the performance overhead of streaming. To address the limitation of one streamed result-set per connection, we are using a pool of connections for streamed result-sets: each streamed result-set requests a connection from the pool when it needs to be computed and returns it back to the pool when it has been fully iterated.

3 Related Work

Several researchers have proposed solutions for translating OCL to SQL. For example, in [BB07], the authors demonstrate an approach for generating event-condition-action (ECA) rules comprising SQL triggers and procedures from OCL constraints attached to a UML class diagram, when the latter is translated into a relational schema. In [DHL01], the authors propose using OCL-derived views in relational databases designed using UML, to check the integrity of the persisted data. In this work each OCL constraint is translated into a view in the relational database that contains reports of integrity violations. Such violations can be handled using different strategies including rolling back the offending transaction, triggering a data reconciliation action, or simply reporting the violation to application users. This approach has been implemented in the context of the OCL2SQL prototype[5]. A similar approach is proposed by the authors of [U. 09]. In [Hei07], the authors propose a framework for translating OCL invariants into multiple query languages including SQL and XQuery using model-to-text transformations.

All the approaches above propose compile-time translation of OCL to SQL. By contrast, our approach proposes run-time generation and lazy evaluation of SQL statements. While compile-time translation is feasible for side-effect free OCL constraints that are evaluated against a homogeneous target (e.g. a relational database), for use-cases that involve querying and modifying models conforming to different technologies (e.g. a relational database and an EMF model), this approach is not applicable. Another novelty of the approach proposed in this paper is that it does not require a UML model that specifies the schema of the database, and as such, it can be used on existing databases that have not been developed in a UML-driven manner.

[5] `http://dresden-ocl.sourceforge.net/usage/ocl22sql/`

4 Conclusions and Further Work

In this paper we have argued that it is important for model management languages to extend their scope beyond the narrow boundaries of 3-level metamodelling architectures such as MOF and EMF. In this direction, we have experimented with using an OCL-based imperative transformation language to query data stored in relational databases. We have reported on the identified challenges and proposed an approach for improving the performance of some types of queries using run-time query translation.

In future iterations of this work, we plan to investigate the extent to which compile-time static analysis and query rewriting can deliver additional benefits in terms of performance. An obvious target is to use static analysis to limit the number of columns returned by queries by excluding any columns that are never accessed in the model management program, but additional optimisations are also envisioned. We also plan to investigate supporting queries spanning more than one tables by exploiting foreign keys.

References

[BB07] D. Berrabah and F. Boufares. Constraints checking in uml class diagrams: Sql vs ocl. In Roland Wagner, Norman Revell, and Ganther Pernul, editors, *Database and Expert Systems Applications*, volume 4653 of *Lecture Notes in Computer Science*, pages 593–602. Springer Berlin Heidelberg, 2007. URL: `http://dx.doi.org/10.1007/978-3-540-74469-6_58`, `doi:10.1007/978-3-540-74469-6_58`.

[DHL01] Birgit Demuth, Heinrich Hussmann, and Sten Loecher. Ocl as a specification language for business rules in database applications. In *Proc. 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML '01, pages 104–117, London, UK, UK, 2001. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=647245.719455`.

[Dim06] Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.

[Dim12] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, James Williams, Richard F. Paige. A Lightweight Approach for Managing XML Documents with MDE Languages. In *Proc. 8th European Conference on Modeling Foundations and Applications*, Copenhagen, Denmark, July 2012.

[Hei07] Heidenreich, F. and Wende, C. and Demuth, B. A Framework for Generating Query Language Code from OCL Invariants. *Electronic Communication of the European Association of Software Science and Technology*, 9, 2007.

[Mar13] Martins Francis, Dimitrios S. Kolovos, Nicholas Matragkas, Richard F. Paige. Adding Spreadsheets to the MDE Toolbox. In *Proc. ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Miami, USA, October 2013.

[Ric09] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack. The Design of a Conceptual Framework and Technical

Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009.

[U. 09] U. Marder, N. Ritter, H.-P. Steiert. A DBMS-based Approach for Automatic Checking of OCL Constraints. In *Proc. "Rigourous Modeling and Analysis with the UML: Challenges and Limitations, OOPSLA workshop*, 2009.

About the authors

Ran Wei is an associate professor at the Dalian University of Technology, China. Contact him at `r.wei@live.co.uk`.

Konstantinos Barmpis is a researcher of the Department of Computer Science at the University of York, United Kingdom. Contact him at `konstantinos.barmpis@york.ac.uk`.

Dimitrios Kolovos is a professor of the Department of Computer Science at the University of York, United Kingdom. Contact him at `dimitris.kolovos@york.ac.uk`.

Acknowledgments This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the Automated Measurement and Analysis of Open Source Software (OSSMETER) FP7 STREP project (318736).