ELM to SQL Translation

# Overview

This document describes an approach to translation of ELM to SQL utilizing the HeD Schema Framework. The document describes changes made to the HeD Schema Framework to support type verification of ELM libraries, as well as the addition of an SQL translation extension to support expression of ELM libraries as a series of SQL views.

The translation utilizes as much as possible the existing HeD Schema Framework. For representation and construction of the resulting SQL, the approach makes use of some libraries from an open source Federated DBMS called Alphora Dataphor.

The approach is based on T-SQL, the dialect used by Microsoft SQL Server. However, dialect-specific aspects of the translation are documented in the source code, and the libraries used support implementation of translation to any dialect, so the approach can be easily modified to support other target systems such as Oracle, MySQL, or PostgreSQL.

# ELM Type Verification

The first step in the approach was to update the HeD Schema Framework to be able to perform type verification on libraries expressed in the Expression Logical Model (ELM), the shareable representation of clinical quality logic defined by the Clinical Quality Language specification.

Because the ELM has a strong heritage from HeD expressions, there is a one-to-one correspondence between most of the operations represented within ELM and an equivalent HeD representation. As such, many of the type verifiers used within HeD can be used directly by the ELM translator. For operations that are new in ELM, new verifiers were built, and a verification handlers map for ELM was built to define the handlers for each ELM expression type.

In addition, because the ELM output for the Chlamydia screening artifacts being translated are based on the FHIR Quality Profiles (QUICK), the HeD Schema Framework was extended to include a FHIR.Model assembly to represent the data structures of FHIR.

By combining these two extensions with the existing HeD Schema Framework, we can now perform type verification on ELM libraries output by the reference implementation of the CQL-to-ELM translator.

# SQL Translation

The next step involved the construction of an SQL translator for the HeD Schema Framework. The approach taken by the translator is to produce SQL view definitions for each ExpressionDef present in the ELM library.

## Overview

The approach taken is to translate all top-level expressions defined in the libraries being translated to SQL view definition statements. Because all the views will end up in the same database, a naming convention including the library or artifact name of the ELM definition is used. For example, the Chlamydia Screening library is named *ChlamydiaScreeningCDS* and defines a top-level expression named *Patient*. This results in an SQL view named *ChlamydiaScreeningCDS\_Patient* in the resulting output.

In addition, because the type of an SQL view must always be a table, but the type of a named expression within ELM may be any type, the translation process ensures that a non-list-valued expression definition in ELM is promoted to a table when it is translated to the view definition, and demoted back to the appropriate type when it is referenced within other expressions. For example, the result type of the *Patient* expression in ELM is a tuple with elements corresponding to the attributes of the Patient record. In SQL, this is promoted to a table using a *select* expression, and when referenced within expressions, it is demoted to the appropriate type using subqueries.

The translation proceeds by converting each ELM expression definition into an equivalent SQL representation, and then outputting the result as a text file containing SQL view definition statements.

The translation currently does not perform any transformation of the data model involved, other than the basic transformation of primitive types to SQL-based equivalents. Because the ELM artifacts are using the FHIR Quality Profiles (QUICK), this approach requires that the SQL be accessible in this way. This issue is discussed further in the section Representation of FHIR in SQL-based Systems.

## Representation of SQL

The translator constructs SQL using language representation components from an open source Federated Database Management System called Alphora Dataphor. The Alphora.Dataphor.Language.SQL assembly contains classes for the representation of all major SQL dialects, as well as emitters for producing the actual SQL based on those representations.

As noted in the overview, the approach taken here produces T-SQL, the SQL dialect used by SQL Server as well as Sybase systems. However, because of the way the language representation facilities are built, most of the logic is common to all dialects, and only a few places are T-SQL specific. These places are noted within the translation code, and the effort required to produce translations for different dialects is minimal, involving changing how those dialect specific translations occur.

The following table describes the major classes used to represent SQL:

|  |  |
| --- | --- |
| Class | Description |
| Statement | Abstract base class for the representation of any statement of a language. |
| Expression | Abstract base class for the representation of any expression of a language. |
| UnaryExpression | Represents a unary expression (such as logical or arithmetic negation). |
| BinaryExpression | Represents a binary expression (such as logical conjunction or addition) |
| ValueExpression | Represents a literal (such as an integer or string value). |
| SelectExpression | Represents a simple select-from-where query. |
| QueryExpression | Represents a query containing table operations such as union and intersect. |
| SelectStatement | Represents a full query statement. |

For more information, see the classes defined in the Language, Language.SQL and Language.TSQL namespaces.

The translators described in the next section work by building instances of these and other classes to represent the equivalent SQL expression.

## Translators

The HeD Schema Framework provides a basis for implementing translation by providing translation classes for each type of expression. The translation proceeds by starting at the root of the expression representation in ELM, and constructing the appropriate type of translator for each node of the tree. The translator is then responsible for constructing and returning the equivalent representation for the type of node.

In addition, the translation extends the base TranslationContext provided by the HeD Schema Framework with an SQLTranslationContext that provides additional services for managing the translation process. In particular, the context provides the naming service used to map the name of an object reference within ELM to the equivalent name within the translated SQL.

The following sections discuss the translation performed for the various types of expressions within ELM.

### Literals

When literals appear in an ELM expression, they are translated to an equivalent SQL representation of the literal value. For most primitive types, this is simply a direct translation (e.g. the integer value 1 in ELM is directly expressed as the integer value 1 in SQL). However, for primitive types, there are two exceptions: Boolean, and DateTime.

SQL does not have a native representation for Boolean values, Booleans can only be used as the result of Boolean-Valued expressions. To address this issue, the translator uses the integer values 0 and 1 to represent *false* and *true*, respectively. A Boolean literal is then represented in SQL as a comparison expression against the integer representation. For example, the Boolean literal *true* in ELM is represented as *1 = 1* in SQL.

For DateTime values, the T-SQL dialect does not have a specific constructor, relying instead on implicit conversion from appropriately formatted string values. To avoid any potential issues with implicit conversion, the translator uses an explicit *Convert* to translate DateTime literals. For example, the value *DateTime(2014, 12, 10)* is represented in T-SQL as *Convert(datetime, ‘2014-12-10’)*.

For Interval and List values, the translation assumes that the target environment contains appropriate user-defined types and functions for representing and manipulating interval values.

Tuple values in ELM are represented as rows in the resulting SQL.

### Expressions

In most cases, the translation of expressions involves simply determining the appropriate name of the operator to use in the target environment. For example, the *Add* operator in ELM is represented using the standard plus (*+*) operator. The translator currently supports most of the basic expression operators, along with enough of the list operators to achieve a translation of the Chlamydia Screening library. In particular, the *In* operator is supported, and is represented in SQL as an *exists* subquery for non-scalar-valued elements, and as an *in* subquery for scalar-valued elements.

*SingletonFrom*, *First*, and *Last* are also supported, as translations to an appropriate *select* statement with a *top* clause. Note that this is a T-SQL-specific solution, other dialects use different syntaxes to represent quota queries. The conceptual translation of these operators would be the same, just using a different syntax.

### Queries

Query expressions within ELM can have many different components, resulting in expressions of varying complexity. Translation of Query expressions is performed by mapping the functionality of each component of an ELM query to its equivalent clause in an SQL *select* statement.

#### Query Source

The source for an ELM query may be any number of expressions. For each expression, the resulting SQL *select* statement will have a *table specifier* in the *from* clause with the equivalent SQL for that expression. In the simplest case, this is just a reference to a table. For example, the query:

[Patient] P

would be represented as:

select \* from Patient P

If the query involves a reference to a named expression:

Patient P

Then the naming convention for named expressions is used, and the query is represented as:

select \* from LibraryName\_Patient P

If the query involves multiple sources, each source is listed in the from clause of the resulting select:

Patient P, MedicationStatement S

would be represented as:

select \* from Patient P, MedicationStatement S

#### Define Clause

The Define clause within ELM is represented in SQL using nested queries. For example:

Encounter E define lengthOfStay = duration in days of E.period

would be represented in SQL as:

select \*   
 from   
 (  
 select \*, lengthOfStay = DateDiff(day, E.period.low, E.period.high)   
 from Library\_Encounter E  
 ) E

This is necessary in order to allow the *lengthOfStay* result to be referenced by other clauses within the query.

Note that define clauses are not yet implemented in the SQL translator.

#### Relationship Clause

Relationship clauses are represented in SQL as an *exists* or *not exists*, depending on whether the relationship is a *with* or *without*, respectively. For example:

Encounters E with Conditions C such that C.onsetDateTime >= E.period.low

would be represented as

select \* from Encounters E   
 where exists (select \* from Conditions C where C.onsetDateTime >= E.period.low)

#### Where Clause

Where clauses are represented in SQL as a *where* clause. For example:

Encounters E where E.period.low >= DateTime(2014, 1, 1)

would be represented as:

select \* from Encounters E where E.period.low >= convert(datetime, ‘2014-01-01’)

#### Return Clause

The return clause is represented in SQL using the column list of the *select* statement. For example:

Encounters E return tuple { id: E.identifier, lengthOfStay: duration in days of E.period }

would be represented in SQL as:

select E.identifier as id, DateDiff(day, E.period.low, E.period.high) as lengthOfStay  
 from Encounters E

#### Sort Clause

And finally, the sort clause is represented in SQL using the *order by* clause of the *select* statement:

Encounters E sort by period.low

would be represented in SQL as:

select \* from Encounters E order by period.low

### ExpressionDef/ExpressionRef

As described in the approach, translation of an ExpressionDef produces a *create view* statement in the output SQL. If the type of the expression in ELM is list-valued, the create view statement just contains the translated expression. If the type is non-list-valued, the create view statement will promote the expression to a query, and the ExpressionRef translation will “dereference” the view. For example, many of the expression definitions in the Chlamydia Screening library are simply Boolean-valued expressions such as DocumentedSexuallyActive, and HasHadConditionOfSexualActivity. These expressions are represented in SQL by embedding the expression in a select statement with a single row and column, and naming the column *value*. For example:

define DocumentedSexuallyActive = exists ([Condition: Sexually Active])

would be represented in SQL as:

create view ChlamydiaScreeningCDS\_DocumentedSexuallyActive as  
 select case when exists (select \* from [Condition] T where ...) then 1 else 0 end as value

When this expression is referenced within another expression via an *ExpressionRef*, then it must be “demoted” back to a boolean value, which is accomplished using a subquery:

DocumentedSexuallyActive or HasHadConditionOfSexualActivity

would be represented as:

(select value from ChlamydiaScreeningCDS\_DocumentedSexuallyActive) = 1  
 or (select value from ChlamydiaScreeningCDS\_HasHadConditionOfSexualActivity)

Note that these examples are boolean-valued and so show the extra scaffolding for representing boolean values as integers within the SQL. If the expression definitions in ELM were integer-valued, for example, the extra scaffolding would not be required:

define PatientBirthDate = (singleton from [Patient]).birthDate

Would be represented as:

create view LibraryName\_PatientBirthDate as  
 select top 1 birthDate as value from Patient

And an expression referencing it would be:

PatientBirthDate >= DateTime(2000, 1, 1)

(select value from LibraryName\_PatientBirthDate) >= convert(datetime, ‘2000-01-01’)

### Outstanding Items

The translation is currently sufficient to support translation of most of the Chlamydia Screening artifact. Most of the outstanding work is in filling out translators for additional functionality. The file TranslationHandlerMap.xml in the SQL.Translation project details the translators that have been built. All the node types are present in that file, but nodes that are not yet built (or not yet functional) have been commented out, so this file provides a checklist for producing a fully featured SQL translator for ELM.

## Translation Output

Once the translators have produced an in-memory representation of the view definitions, these are fed to an Emitter to produce the actual text of the SQL, and this text is saved to a file in the same directory as the input file.

# Representation of FHIR in SQL-based Systems

The approach taken here to the representation of FHIR data is the simplest approach from the perspective of the translation effort, in that it allows the translation to assume that the data in SQL “looks” exactly like the native FHIR data. This approach may prove untenable for the following reasons:

1. The approach may be too complex, involving a significant effort to produce SQL views that expose data from a particular model in this way.
2. The approach may not be performant, involving a significant computational cost to transform data to the expected views.

However, as a first pass, the approach is useful in demonstrating the ability to translate ELM to SQL. Several possible mitigations for these potential issues exist, including:

1. Incorporating model transformation into the SQL translator, allowing the output SQL to be executed either directly against the source data model, or against an intermediate that is “flatter” than FHIR.
2. “Materialization” of the FHIR views, i.e. rather than having view definitions, create equivalent table structures and populate them from the source data model using a series of transformation procedures. This approach is tenable for the Quality Measurement scenario where all data is historical. However, it may not be tenable for a Clinical Decision Support application.

## Data Types

For truly primitive types, such as Integers, Strings, and DateTimes, the translation uses a direct mapping from the XSD type (xs:integer, xs:string, xs:datetime, etc.) to the equivalent SQL type. The only exception being xs:boolean, which as discussed above must be represented in SQL using integers in literal contexts.

However, FHIR “primitive” types are not strictly speaking primitive types, in that they derive from Element, and as such have additional attributes such as “id” and “extension”. As a result, they are modeled in this approach using user-defined types that contain the attributes available in SQL. For example, the FHIR Integer type would be modeled using a user-defined data type defined as follows:

FHIRInteger  
{  
 id: string  
 value: integer  
 extension: List<Extension>  
}

The CQL-to-ELM translator is aware of this structure, and implicitly produces accessors as necessary. For example, when comparing an integer-valued attribute from a FHIR structure:

Patient.age >= 18

The actual ELM is output with an additional property accessor:

Patient.age.value >= 18

And the user-defined types defined above would then work as expected for the resulting SQL representation:

select \* from Patient where age.value >= 18

Beyond the primitive types, any types that must appear as elements within other types, the target environment must have a user-defined type to support the embedding of that value within another value.

In addition to the ability to represent types, FHIR often uses the type-inheritance capabilities of XML to indicate that an element may be of any subtype of a given type. For example, the Resource element of the ResourceContainer. Where these relationships are modeled in XML with a choice, the resulting types are expected to expose an attribute for each possible type. For example:

class FHIRResourceContainer  
{  
 alert: FHIRAlert,  
 allergyIntolerance: FHIRAllergyIntolerance,  
 ...  
}

## Lists

Lists are used throughout the FHIR structures. For systems that provide native support for array or collection types such as PostgreSQL and Oracle, the native support would be used. However, T-SQL does not provide such support, and lists must therefore be represented as user-defined types. This approach would then require that every type that must be able to appear in a list would have a List equivalent, such as “FHIRintegerList”, “FHIRCodeableConceptList”, and so on.

## Resource Types

In general, resource types are represented as tuple values within ELM, and end up being represented as tables within the SQL, where every element in the FHIR type is a column in the table of the appropriate user-defined type. For example, the following SQL table represents the Patient FHIR data structure:

create table Patient  
(  
 -- from Resource  
 id FHIRid,  
 meta FHIRResourceMeta,  
 language FHIRcode,  
 -- from DomainResource  
 text FHIRNarrative,  
 contained FHIRResourceContainerList,  
 extension FHIRExtensionList,  
 modifierExtension FHIRExtensionList  
 -- from Patient  
 identifier: FHIRIdentifierList,  
 name: FHIRHumanNameList,  
 telecom: FHIRContactPointList,  
 gender: FHIRAdministrativeGender,  
 birthDate: FHIRdateTime,  
 deceasedBoolean: FHIRboolean,  
 deceasedDateTime: FHIRdateTime,  
 address: FHIRAddressList,  
 maritalStatus: FHIRCodeableConcept,  
 multipleBirthBoolean: FHIRboolean,  
 multipleBirthInteger: FHIRinteger,  
 photo: FHIRAttachmentList,  
 contact: FHIRPatientContactList,  
 animal: FHIRPatientAnimal,  
 communication: FHIRCodeableConceptList,  
 careProvider: FHIRReferenceList,  
 managingOrganization: FHIRReference,  
 link: FHIRPatientLinkList,  
 active: FHIRboolean,  
)

## Conclusions

Clearly, the approach described above involves a significant effort, both in producing the required user-defined type and operators, and in producing the transformation logic from the source data model to the resulting FHIR definitions. One immediately simplifying prospect would be to only provide support for attributes that are actually present. For example, the “id” and “extension” attributes of the FHIR “primitive” types could potentially be ignored. In addition, many of the Patient attributes described above can be safely ignored, especially for the specific application of Chlamydia Screening.

The recommendation at this point is to move forward with a very basic implementation of this representation, stripped down to only the Patient, Condition, and whatever attributes of those resources are actually referenced by the Chlamydia Screening artifact, and use these as a basis for investigation of the approach.