PTC

Design Note

Translating QBATI Filters to SQL WhereExpressions

Jeff Clark

02/05/14

Summary:

QueryBasedAssociationTypeIdentifiers (QBATIs) are defined in the paper “Multi-language support for attribute values in Windchill: Approach”, by Joe McKinley, March, 26, 2013.

This paper is a companion to the Design Note titled “Association Type Identifier Role Processing”, by Jeff Clark, which describes the parsing of a QBATI “role filter” string into a parse tree representation and the generation of an AttributeContainerSet representation from that tree.

This paper describes the implementation of the Basic Query Service “result handler” that processes QBATIs encountered during the generation of a SQL query. The result handler performs the translation of the AttributeContainerSet representation of a QBATI filter into an equivalent SQL WhereExpression. These WhereExpressions specify SQL table joins used to associate objects of the QBATI “context” type with objects of the QBATI “association” type.

Background

A QueryBasedAssociationTypeIdentifier is an object which specifies relational criteria that associate an instance of one type with an instance of another type, based on potentially complex Boolean expressions involving attributes defined by the two types. The initial expected use of QBATIs is to retrieve *Translated Text* values for attributes of a type for which translations have been provided, based on the locale of the user.

An example QBATI with a role filter is:

```
WCTYPE|wt.part.WTPart~QBA|
[(foo=masterText)AND(translationInfo.authoringLanguage=sourceLanguage)AND(123456789=d
ictionaryReference.key.id)AND('en_us'=targetLanguage)AND(thePersistInfo.modifyStamp>e
ffectiveStart)AND((NULL=effectiveEnd)OR(thePersistInfo.modifyStamp<=effectiveEnd))]
@WCTYPE|com.ptc.core.td.server.dictionary.TranslationDictionaryEntry
```

The role filter is the text contained between the matching "[ ]" characters. (Note that the above example is not completely valid, as there is no "foo" attribute defined for WTPart.)

The subexpressions in the QBATI specify criteria that are used to "join" the rows of the "context" type (WTPart) with the rows of the "data" (or "association") type (TranslationDictionaryEntry). Some of the expressions in the "role filter" specify column values to be "matched" between the two tables (i.e., the "join" criteria), while others specify limits on the acceptable rows of the "data" type that may participate in those joins. For example, the subexpression (foo=masterText) specifies that the WTPart and TranslationDictionaryEntry tables will be joined based on the columns containing their "foo" and "masterText" attributes, respectively.

The conditions that join the rows of the context type and the association type may be satisfied by multiple rows. A QBATI may contain additional conditions to filter these results. For example, the subexpression ('en_us'=targetLanguage) specifies that the columns that are selected from the TranslationDictionaryEntry table by the join operation must have the value 'en_us' in their "targetLanguage" column. The remainder of the conditions in the above example further restrict the allowable set of returned rows from the TranslationDictionaryEntry table to ensure that only a single row is selected from that table for each row from WTPart.

Basic Query Service Handlers

The existing Basic Query Service infrastructure contains mechanisms that will invoke configured "handlers" to process different types of TypeIdentifiers when constructing a SQL query statement. There are (at least) two different types of handlers: criteria and result. For QBATIs, we only need a result handler. There was considerable discussion as to whether or not a "criteria handler" was also needed to support the use of QBATIs, particularly by the Search team. The requirement for Search is to find all the parts where the value of an attribute is equal to some specified value, i.e., where an attribute is specified with a QBATI as the context in a CriteriaSpec.

According to Joe Thompson, the result handler infrastructure supports returning the value of an attribute specified by a QBATI for a given object, i.e., "... the QBATI ResultHandler implementation will satisfy it. The soft query layer already uses ResultHandler implementations to handle the context of MBA criteria."

Based on these assertions, we do not need to implement QBATI CriteriaHandler to satisfy the Search team's requirements.

QueryBasedAssociationResultHandler

We will create a new class, QueryBasedAssociationResultHandler, that extends the existing class AbstractAssociationResultHandler. The new class will be registered as the "result handler" for QBATIs in Windchill/DevModules/Foundation/src/dbservice.properties.xconf, in the section for com.ptc.core.query.server.impl.ResultHandler, like this:

```
<Option cardinality=”duplicate”
   requestor=”com.ptc.core.meta.common.impl.QueryBasedAssociationTypeIdentifier”
  selector=”ATTRIBUTE”
  serviceClass=”com.ptc.core.query.server.impl.QueryBasedAssociationResultHandler”/>
```

There are two methods that will be implemented in QueryBasedAssociationResultHandler that are defined in the abstract parent:

```
protected Set <AssociationTypeIdentifier> getAdditionalContexts()

protected abstract WhereExpression buildWhere (Map <TypeIdentifier, FromClassIndexInfo>
a_contextInfoMap) throws WTException;
```

The implementations of these methods in our new class will call "getDefinition()" (inherited from the parent class) to get the QBATI to be translated and invoke its "getFilter()" API to obtain the AttributeContainerSet for that QBATIinstance.

The getAdditionalContexts() implementation will assemble the Set of TypeIdentifiers for attributes that are referenced by the QBATI filter string that are *not* defined on the context type itself, but rather are defined on a type that is associated to the context type, for example, auto-nagivate attributes defined on WTPartMaster but which are referenced from WTPart.

The buildWhere() implementation will generate a WhereExpression from the obtained AttributeContainerSet.

Handling Context Attributes Defined on Associated Types

A typical QBATI like this ...

```
WCTYPE|wt.part.WTPart~QBA|
[(foo=masterText)AND(translationInfo.authoringLanguage=sourceLanguage)AND(123456789=d
ictionaryReference.key.id)AND('en_us'=targetLanguage)AND(thePersistInfo.modifyStamp>e
ffectiveStart)AND((NULL=effectiveEnd)OR(thePersistInfo.modifyStamp<=effectiveEnd))]
@WCTYPE|com.ptc.core.td.server.dictionary.TranslationDictionaryEntry
```

... appears to reference only attributes defined on WTPart and TranslationDictionaryEntry. However, 'translationInfo.authoringLanguage' is actually defined on WTPartMaster (although it is an "auto navigable attribute" accessible from WTPart).

The result handler infrastructure relies upon the Set <AssociationTypeIdentifier> returned by getAdditionalContexts() to assist the buildWhere() API in resolving these references. The getAdditionalContexts() API must generate a Map of TypeIdentifiers → FromClassIndexInfo, which will be passed to the buildWhere() API when it is invoked by the infrastructure.

This allows buildWhere() to determine the appropriate “from index” for each of the attributes referenced by the SearchConditions it generates as part of the resulting WhereExpression.

The implementation of getAdditionalContexts() in QueryBasedAssociationResultHandler must traverse the AttributeContainerSet that is produced from the QBATI's filter and add the TypeIdentifiers of any operands of its functions to the returned set of TypeIdentifiers, excluding the QBATI's 'context' and 'association' types. The Basic Query Service result handler infrastructure will generate “from indices” for all of these types (including the explicit 'context' and 'association' types) and pass them to the result handler's buildWhere() API.

Generating WhereExpressions

The AttributeContainerSet obtained from a QBATI filter by QueryBasedAssociationResultHandler is, essentially, a tree. Each node of the tree is either a CompositeAttributeContainerSet or a SingleCriterionAttributeContainerSet.

CompositeAttributeContainerSets always represent a set of criteria that are to be ANDed or ORed together. The criteria within these sets are either more CompositeAttributeContainerSets or SingleCriterionAttributeContainerSets.

Each SingleCriterionAttributeContainerSet is a wrapper containing one BooleanAttributeContainer-Function (BACF). The BACF represents a comparison operation that yields a Boolean result. Most BACFs take two operands. These operands are either:

- Two attributes, one from the “context” type and one from the “data” type (or “association” type). These types are defined in the QBATI and are also provided to the buildWhere() API as arguments.
- One attribute and one constant value. In this case, the attribute will be defined in the “data” (or “association”) type and the constant is guaranteed to be of an appropriate type for that attribute. That is, if the attribute is of type Boolean, the constant value will be “TRUE” or “FALSE”. If the attribute is an Integer (Long), the constant will be an integer constant value.

Some BACFs take only a single operand, e.g., the ACFIsNull and ACFNot functions. The ACFIsNull function is used to determine if the value stored in the database for the specified attribute is *null* for a given object instance. The ACFNot function is used to negate the (Boolean) result of its operand, which is always another BooleanAttributeContainerFunction.

Since the AttributeContainerSet that is obtained by the getFilter() API is a tree, generating a WhereExpression equivalent to the tree will be performed by a simple recursive “walk” of the tree in “post-order”. As each node is “visited” in the tree walk, we will first process the node's left child, then process the node's right child, then apply the operation indicated by the current node to the results obtained for its left and right children.

The processing applied to nodes that represent object attributes will return a String whose value is the name of the database table column holding that attribute for the appropriate type. The processing for a constant value will be to return an object of the appropriate type for that value, i.e., a String, Long, or Boolean. (The “NULL” constant value that appears in QBATI role filters is handled separately and will not appear explicitly in the AttributeContainerSet tree. It is represented by the AttributeContainerFunction ACFIsNull.)

When processing nodes that represent comparison operations, that is BACFs like ACFEquals or

ACFCompare, we will invoke a utility method appropriate for that specific BACF. These methods are defined in the new class com.ptc.core.query.server.impl.WhereExpressionGenerator. The WhereExpressions that are generated by these utility functions will be blended together into a single WhereExpression as indicated by the 'AND' and 'OR' operations represented by the CompositeAttributeContainerSet nodes in the (AttributeContainerSet) tree.

WhereExpressionGenerator contains several static utility methods that accept a BooleanAttributeContainerFunction as input and return an equivalent WhereExpression for the operation represented by the BACF. Each of these methods is specialized to produce a WhereExpression for a specific subtype of BooleanAttributeContainerFunction. Clearly, this is not an object-oriented design. The “right” way to do this is to add a new API to the BACF interface definition and put these implementations into their respective BooleanAttributeContainerFunction subtypes.

Unfortunately, the types SearchCondition, WhereExpression, etc., are defined in Foundation, while the interfaces and classes related to AttributeContainerFunctions are defined in MetaSpecCommon. Attempting to reference Foundation from MetaSpecCommon would introduce a circular dependency into the build process, so we're stuck with defining a utility class containing type-specific static methods in Foundation.

The WhereExpressionGenerator class will contain the following methods:

- `private static NegatedExpression getNegatedExpression (ACFNot acf)`
- `private static SearchCondition getSearchCondition (ACFIsNull acf)`
- `private static SearchCondition getSearchCondition (ACFCompare acf)`
- `private static SearchCondition getSearchCondition (ACFEquals acf)`
- `private static SearchCondition getSearchCondition (ACFIsElementOf acf)`

This class will also contain one `public` API:

- `public WhereExpression getWhereExpression (BooleanAttributeContainerFunction bacf)`

The `getWhereExpression()` API will use the "instanceof" operator to determine the subtype of the 'bacf' argument and invoke the appropriate BACF subtype-specific API. Since there are other BACFs for which we are not implementing a `getSearchCondition()` method, `getWhereExpression()` will have to be prepared to throw an exception if it is given a BACF for which it does not know how to generate a corresponding WhereExpression.

The advantage of this approach is that the client does not have to differentiate on the subtype of the BooleanAttributeContainerFunction to determine which utility method to call; this detail is hidden within the WhereExpressionGenerator class. This will make the client code that walks the AttributeContainerSet tree and processes each tree node much simpler; it is as close to an object-oriented approach we can get, given the limitations of the existing build module dependencies.

So, instead of obtaining the WhereExpressions for a given ACF like this:

```
WhereExpression we = acf.getWhereExpression ();
```

... we'll do this:

```
WhereExpression we = WhereExpressionGenerator.getWhereExpression (acf);
```

The “from” index values for each operand referenced in a generated SearchCondition must be set appropriately. This can be done directly on the SearchCondition using `setLeftFromIndex()` or `setRightFromIndex()`. Alternatively, it can be done using `WhereExpression.setFromIndicies()`.

For each “leaf” expression in the tree, we must set the “from” index for each attribute. The caller of `WhereExpressionGenerator.getWhereExpression()` is responsible for setting the required “from” index values.

Handling NULL Values

One complication in the generation of WhereExpressions, specifically, SearchConditions arising from Boolean comparisons of two operands, is the handling of *null* values.

For example, when querying for the Translated Text value of an attribute defined in WTPart, there may not be a corresponding TranslationDictionaryEntry defined in the database. Assume we have (simplified) WTPart and TranslationDictionaryEntry tables with the following contents:

WTPart

ida2a2	**Attr1**	**Attr2**	**Attr3**
101	blah	0	foo
102	bletch	1	bar
103	biz	23	boink

TranslationDictionaryEntry

masterText	**TranslatedText**
foo	flitz
bar	bzorp

If we have a QBATI filter fragment that specifies “(Attr3=masterText)”, then this fragment specifies that the tables are to be joined on their “Attr3” and “masterText” columns, respectively. For objects 101 and 102, the corresponding Translated Text values are “flitz” and “bzorp”, respectively.

But for object 103, there are no translations specified for its corresponding Attr3 value. The “normal” join will not match any row of the TranslationDictionaryEntry table with this row of the WTPart table, so the query would return *no row at all* for object 103 (because there is no row in the TranslationDictionaryEntry table that satisfies the join criteria). This is not the desired result when processing a QBATI). A TypeInstance should be returned in the query results for this object even if one of its attribute's context associations does not exist. All other AssociationTypeIdentifiers work this way.

For a row in the WTPart table for which there is no matching entry in the TranslationDictionaryEntry table for the value of the WTPart attribute “Attr3”, we want the query to return the row from the WTPart table, but with *null* as the value for the Translated Text for “Attr3”.

We would normally specify the use of an “Outer Join” on the TranslationDictionaryEntry side of the SearchCondition's expression to obtain this result. That is, for an expression like “Attr3=masterText” (which will be translated into a SearchCondition – a subtype of WhereExpression), we would specify “use outer join” on the right side, while for “masterText=Attr3”, we would specify “use outer join” on the left side. This would be accomplished by:

```
SearchCondition sc = …  // for masterText=Attr3
sc.setOuterJoin (SearchCondition.LEFT_OUTER_JOIN);
```

or

```
SearchCondition sc = …  // for Attr3=masterText
sc.setOuterJoin (SearchCondition.RIGHT_OUTER_JOIN);
```

However, there is a limitation imposed by some databases (Oracle, in particular) which prevents the use of this approach for QBATIs. Outer joins cannot be used in a WhereExpression that includes an

“OR” operator. Since QBATIs *do* allow compound Boolean expressions which include the OR operator, this approach will not work.

There are a couple of ways to work around this. One is to replace the comparison against *null* with a comparison against the value of function that replaces a *null* result obtained from the database with a default value. This would eliminate the chance of getting a *null* result and side-step the need for using outer join. This approach would introduce some rather ugly constructs into the “where expression generation” code: it would have to determine a suitable default value for each possible type (integer, boolean, string, timestamp), compromising the generality of the QBATI mechanism.

An alternative solution, proposed by Joe Thompson, is to ensure that the query service uses a two-pass approach to processing these queries, acquiring the “base attributes” of the context type first, and “filling in” the final result with the attributes obtained by processing the QBATIs in a separate query. Joe states that this mechanism already exists in the query service and is employed in appropriate circumstances. He will ensure that the approach is employed for these queries, as well.

Processing Query Results

AbstractAssociationResultHandler (the parent class of QueryBasedAssociationResultHandler) defines a `process()` method that will create TypeInstances from the raw results returned by the database query. That method is invoked by the Basic Query Service infrastructure when processing query results. It should not be necessary to override that method to process results that have been obtained using the criteria defined in QBATIs.