Database - Bug #2488

replace runtime compilation of dynamic queries

01/14/2015 12:48 AM - Eric Faulhaber

Status:	Closed	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Ovidiu Maxiniuc	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:	Cleanup and Stablization for Server Features		
billable:	No	**case_num:**	
vendor_id:	GCD		

Description

History

#1 - 01/14/2015 12:56 AM - Eric Faulhaber

- Project changed from Liberty to Database

#2 - 01/14/2015 01:26 AM - Eric Faulhaber

Dynamically defined queries are parsed and converted at runtime by P2J, then compiled into discrete Java classes using the InMemoryCompiler. Buffers are set into public instance variables of such a newly compiled class, and in the end, an instance of P2JQuery is extracted from the class via an accessor method. This instance is the P2J representation of the dynamically defined query, which is then used to access database records.

This approach has been a great first step in our dynamic database support and while it appears to work consistently, it unfortunately has some problems. Firstly, its performance is very poor. In code which makes heavy use of dynamically defined queries, the compilation step in particular makes the performance of this solution unacceptable. Profiling (sampling) has shown that most of the CPU time spent during customer testing of such code is spent in InMemoryCompiler.compile and InMemoryFileManager.list (really, in the J2SE classes used by these methods).

Secondly, the implementation of the J2SE compiler which is used under the covers (via javax.tools) causes quite a serious memory leak through its heavy use of soft references and an ever-growing, shared array of characters. This problem can quickly overwhelm the P2J server using default garbage collection settings. To alleviate this leak, we effectively have disabled soft references as a stop-gap measure, but this cannot relied upon as a long term approach for production use.

Thirdly, even with soft references disabled, we still have a huge number of dynamically compiled classes filling up PermGen memory. While each is relatively small, the number of these grows continually into the hundreds of thousands, since the same queries are re-converted and compiled into new classes all the time. This is not a sustainable approach for a long-running server.

This task is about replacing that runtime compilation step with a more performant approach which does not generate any new Java classes at runtime, but which instead analyzes the ASTs generated by runtime conversion, and instantiates queries and supporting objects from standard classes.

#3 - 01/15/2015 02:04 AM - Eric Faulhaber

- Assignee set to Ovidiu Maxiniuc

- File ecf_upd20150115a.zip added

The basic idea behind this improvement is to continue to use runtime TRPL to convert a query to an in-memory, Java AST as we do today, though in a refactored form, and with some enhanced annotations. However, the steps after that will change to eliminate the in-memory compilation.

Today, we generate a JAST which represents a throw-away class that acts essentially as a query cocoon. That class is compiled in memory, instantiated, and its execute method is invoked to produce a single P2JQuery instance. That P2JQuery object is extracted via a getQuery method, then the "cocoon" instance and its associated, newly compiled class is discarded. However, the class definition remains in PermGen memory for the life of the JVM.

Instead of generating a cocoon class JAST and compiling it, we will generate only the essential portions of the JAST which have to do with constructing and initializing the query. During creation of the JAST, we will cache useful annotations for a subsequent tree walk, during which we will invoke Java methods and constructors directly, using reflection. Such annotations will include, for example, a java.lang.reflect.Constructor object for each CONSTRUCTOR JAST node, and a java.lang.reflect.Method object for each METHOD_CALL JAST node. We will have to enhance the com.goldencode.ast.Aast API with a putAnnotation(String, Object) method for this purpose.

To add such annotations, we will have to deduce the appropriate constructor or method at JAST creation time, using the class which is the target of the constructor or method invocation, and the signature represented by the presence, order, and types of the child nodes. Likewise, any references to Java constants, fields, etc. (e.g., LockType.NONE) should be figured out at this stage and annotated, to make it as easy as possible to walk the JAST later and simply assemble argument lists and invoke methods and constructors via reflection. This work of matching methods and constructors based on classes and signatures is not practical to do in TRPL rules, so it probably makes sense to enhance com.goldencode.p2j.uast.JavaPatternWorker$JavaAstHelper with some helper methods to do this work in Java.

There will have to be some mechanism in place to bridge to the state of the running application (like access to the record buffer(s) used in the query). Today, this is done using DynamicQueryHelper$WorkArea.buffers (see the compileQuery method).

At a later date, we could optimize the query conversion phase as well, using caching to eliminate duplication of the most expensive areas of runtime conversion. This will be possible and useful because most applications will duplicate even dynamically defined queries over time. However, this optimization is out of scope for this task.

Today, the JAST is created by the DynamicQueryHelper in two passes. The first pass looks like this for a typical query:

```
compilation unit [COMPILE_UNIT] @0:0
   com.goldencode.p2j.persist.dynquery [KW_PACKAGE] @0:0
   com.goldencode.p2j.util.* [KW_IMPORT] @0:0
   com.goldencode.p2j.util.BlockManager.* [STATIC_IMPORT] @0:0
   InMemP2jQuery205 [KW_CLASS] @0:0
       [CS_CONSTANTS] @0:0
       [CS_STATIC_VARS] @0:0
       [CS_STATIC_INITS] @0:0
       [CS_INSTANCE_VARS] @0:0
         = [ASSIGN] @0:0
            TempRecord1.Buf [REFERENCE_DEF] @0:0
            TemporaryBuffer.define [STATIC_METHOD_CALL] @0:0
               TempRecord1.Buf.class [REFERENCE] @0:0
               ttfiltercriteria [STRING] @0:0
               ttfiltercriteria [STRING] @0:0
                [BOOL_FALSE] @0:0
       [CS_CONSTRUCTORS] @0:0
       [CS_STATIC_METHODS] @0:0
       [CS_INSTANCE_METHODS] @0:0
         execute [METHOD_DEF] @0:0
             [BLOCK] @0:0
               externalProcedure [STATIC_METHOD_CALL] @0:0
                  Block [ANON_CTOR] @0:0
                      [CS_INSTANCE_VARS] @0:0
                      [CS_INSTANCE_METHODS] @0:0
                        body [METHOD_DEF] @0:0
                           block [BLOCK] @0:0
                              buffer_anchor [BOGUS] @0:0
                                 RecordBuffer.openScope [STATIC_METHOD_CALL] @0:0
                                    ttfiltercriteria [REFERENCE] @0:0
                              first [METHOD_CALL] @0:0
                                 FindQuery [CONSTRUCTOR] @0:0
                                    ttfiltercriteria [REFERENCE] @0:0
                                    upper(ttfiltercriteria.fieldname) = 'END-DAT' [STRING] @0:0
                                     [NULL_LITERAL] @0:0
                                    ttfiltercriteria.fieldname asc [STRING] @0:0
                                    LockType.NONE [REFERENCE] @0:0
       [CS_INNER_CLASSES] @0:0
```

```
   com.goldencode.p2j.persist.* [KW_IMPORT] @0:0
   com.customer.app.dmo._temp.* [KW_IMPORT] @0:0
   com.goldencode.p2j.persist.lock.* [KW_IMPORT] @0:0
```

Although we don't anti-parse this form of the JAST, it would look as follows:

```
package com.goldencode.p2j.persist.dynquery;

import com.goldencode.p2j.util.*;
import com.goldencode.p2j.persist.*;
import com.customer.app.dmo._temp.*;
import com.goldencode.p2j.persist.lock.*;

import static com.goldencode.p2j.util.BlockManager.*;

public class InMemP2jQuery205
{
   TempRecord1.Buf ttfiltercriteria = TemporaryBuffer.define(TempRecord1.Buf.class, "ttfiltercriteria", "ttfil
tercriteria", false);

   public void execute()
   {
      externalProcedure(new Block()
      {
         public void body()
         {
            RecordBuffer.openScope(ttfiltercriteria);
            new FindQuery(ttfiltercriteria, "upper(ttfiltercriteria.fieldname) = 'END-DAT'", null, "ttfiltercr
iteria.fieldname asc", LockType.NONE).first();
         }
      });
   }
}
```

The second pass refactors the JAST to null out buffer instance variable definitions and make these instance variables public, remove some Block method cruft placed there by the original conversion rules, and add a getQuery method:

```
compilation unit [COMPILE_UNIT] @0:0
   com.goldencode.p2j.persist.dynquery [KW_PACKAGE] @0:0
   com.goldencode.p2j.util.* [KW_IMPORT] @0:0
   com.goldencode.p2j.util.BlockManager.* [STATIC_IMPORT] @0:0
   InMemP2jQuery205 [KW_CLASS] @0:0
       [CS_CONSTANTS] @0:0
       [CS_STATIC_VARS] @0:0
       [CS_STATIC_INITS] @0:0
       [CS_INSTANCE_VARS] @0:0
         = [ASSIGN] @0:0
            TempRecord1.Buf [REFERENCE_DEF] @0:0
            NULL [NULL_LITERAL] @0:0
         = [ASSIGN] @0:0
            FindQuery [REFERENCE_DEF] @0:0
            NULL [NULL_LITERAL] @0:0
       [CS_CONSTRUCTORS] @0:0
       [CS_STATIC_METHODS] @0:0
       [CS_INSTANCE_METHODS] @0:0
         execute [METHOD_DEF] @0:0
             [BLOCK] @0:0
               = [ASSIGN] @0:0
                  findQuery [REFERENCE] @0:0
                  FindQuery [CONSTRUCTOR] @0:0
                     ttfiltercriteria [REFERENCE] @0:0
                     upper(ttfiltercriteria.fieldname) = 'END-DAT' [STRING] @0:0
                      [NULL_LITERAL] @0:0
                     ttfiltercriteria.fieldname asc [STRING] @0:0
                     LockType.NONE [REFERENCE] @0:0
         getQuery [METHOD_DEF] @0:0
            block [BLOCK] @0:0
```

```
                [KW_RETURN] @0:0
                  findQuery [REFERENCE] @0:0
       [CS_INNER_CLASSES] @0:0
   com.goldencode.p2j.persist.* [KW_IMPORT] @0:0
   com.customer.app.dmo._temp.* [KW_IMPORT] @0:0
   com.goldencode.p2j.persist.lock.* [KW_IMPORT] @0:0
```

...and the anti-parsed form:

```
package com.goldencode.p2j.persist.dynquery;

import com.goldencode.p2j.util.*;
import com.goldencode.p2j.persist.*;
import com.customer.app.dmo._temp.*;
import com.goldencode.p2j.persist.lock.*;

import static com.goldencode.p2j.util.BlockManager.*;

public class InMemP2jQuery205
{
   public TempRecord1.Buf ttfiltercriteria = null;

   FindQuery findQuery = null;

   public void execute()
   {
      findQuery = new FindQuery(ttfiltercriteria, "upper(ttfiltercriteria.fieldname) = 'END-DAT'", null, "ttfi
ltercriteria.fieldname asc", LockType.NONE);
   }

   public P2JQuery getQuery()
   {
      return findQuery;
   }
}
```

What we really need is a bare bones JAST without all the Java class structure cruft, which is just about the construction and configuration of the query, plus any initialization (e.g., database alias creation) which might be needed to prepare the query.

I'm not sure yet whether it makes sense to do the final JAST walk in TRPL or Java code.

I'm also not 100% clear on the design of the bridge mechanism to make the buffer objects available as arguments to the query constructors or methods. A hash map seems to be a good way to relate the name of the buffer in a REFERENCE node to the actual buffer proxy object in memory.

The attached version of DynamicQueryHelper writes the above AST and Java source output to stdout.log. You may find it useful.

Please ask questions and document any ideas you may have on this topic. I want to make sure we have a common understanding of our approach and design.

#4 - 01/15/2015 06:41 AM - Ovidiu Maxiniuc

I understand the principle of this approach.

At first sight, it looks rather easy, the where clause and sorting are already computed as strings. The reflection should be able to detect the appropriate constructor for FindQuery using the arguments stored in an Object[] array.

The difficulty appears when there are another buffers (and aliases) and even more, when some client-side where clause has to be constructed also. We should do some tests to detect the maximum complexity of these and find a solution to instantiate this object.

The java/TRLP debate. It's clear that TRPL has the some advantages, but I think that we should prefer java processing at least because of the additional speed gained because the code is already compiled and the direct manipulation of objects.

#5 - 01/15/2015 12:41 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> The difficulty appears when there are another buffers (and aliases)...

Please explain your concerns in detail on both these points.

> ...and even more, when some client-side where clause has to be constructed also. We should do some tests to detect the maximum complexity of these and find a solution to instantiate this object.

Yes, we have to design for the most complex circumstances possible, since if it is possible, someone will have done it. I do agree that the client-side where clause processing is somewhat problematic, due to it being a callback and the open-ended nature of the expressions that can be represented there.

At the core of this approach, I see something like a Runnable interface, but which returns the object instance we care about. Perhaps something like this:

```
public interface Interpretable<T>
{
   public T interpret(Aast jast);
}
```

The idea being that T is the type of object you want returned, like a P2JQuery or a WhereExpression, and the implementation of interpret(Aast) is where the walk happens.

This idea is not fully formed yet, so I welcome feedback.

I'm not sure how well an interpreted WhereExpression would work. We may still have to fall back to a compiled approach for these, since the expressions they represent can be quite flexible, though I'd like to avoid that if possible. The good news, however, is that they should be relatively rare; at least, that's what we've found with static conversion.

#6 - 01/15/2015 12:47 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> The java/TRLP debate. It's clear that TRPL has the some advantages, but I think that we should prefer java processing at least because of the additional speed gained because the code is already compiled and the direct manipulation of objects.

I tend to agree in terms of the final walk of the JAST, where we actually invoke the constructors and methods using reflection. Although TRPL is designed for walking ASTs, I think in this case the advantages of using Java for this stage of the process outweigh those of TRPL.

However, I think all of the other steps up to that point to create the JAST should continue to be in TRPL, with callouts to helper methods from JavaPatternWorker (or a new pattern worker if they're extensive) for the messy parts (e.g., figuring out which c'tor or method variant is the right one for a particular JAST node, etc.).

#7 - 01/15/2015 01:05 PM - Ovidiu Maxiniuc

> I'm not sure how well an interpreted WhereExpression would work. We may still have to fall back to a compiled approach for these, since the expressions they represent can be quite flexible, though I'd like to avoid that if possible. The good news, however, is that they should be relatively rare; at least, that's what we've found with static conversion.

In #2355 I fixed a defect with these expressions, I am trying now to determine how complex these might be on the customer server project (as a minimum requirement for now).
On the other hand, the QUERY-PREPARE only accept only quoted constants or an unabbreviated, unambiguous buffer/field reference for buffers known to query otherwise error 7328 will occur. The predicate of FIND-FIRST/LAST methods has a fixed syntax that might reduce the complexity.

> However, I think all of the other steps up to that point to create the JAST should continue to be in TRPL, with callouts to helper methods from JavaPatternWorker (or a new pattern worker if they're extensive) for the messy parts (e.g., figuring out which c'tor or method variant is the right one for a particular JAST node, etc.).

True. In the end, if this step is still a bottleneck, creating the JAST tree is the possible subject for further optimization.

#8 - 01/15/2015 02:34 PM - Eric Faulhaber

I think the following would give us a reasonably complex WhereExpression:

```
find first <permanent table buffer>
where
  <permanent table buffer>.<integer field> = <integer constant>
and
  if <local logical variable>
     then true
     else can-find(first <temp-table buffer> where <temp-table buffer>.<integer field> = <integer constant>)
```

I haven't actually tried this; is it possible to do a dynamic find-first() that is the equivalent of this construct, or would this trigger a Progress error?

#9 - 01/15/2015 02:42 PM - Ovidiu Maxiniuc

Looking at find-first() method in manual, the predicate expression is very simple, just

```
[ WHERE [ logical-expression ] ] [ USE-INDEX index-name ]
```

It can contain only constants and unabbreviated references to fields from **the buffer**.
I understand that these are really basic expressions.

#10 - 01/15/2015 02:53 PM - Eric Faulhaber

OK, so no calls to UDFs, either? That's another way to trigger client-side where clause processing.

If not, it would seem it is not possible to generate a client-side WhereExpression from a find-*() method. That's good news.

Is it possible to do either of these things in a query-prepare() predicate expression? The documentation makes it seem like those are more open.

#11 - 01/15/2015 02:58 PM - Ovidiu Maxiniuc

Yes, query-prepare() is the big issue. I believe there are no constrains here and any number of buffers can be added using set/add-buffer() methods.
I will try to construct an example based on your note #8 above.
I good idea is to log the encountered queries from the server itself while running the set of tests. That way we should get an idea of what we should expect.

#12 - 01/15/2015 03:12 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Yes, query-prepare() is the big issue. I believe there are no constrains here and any number of buffers can be added using set/add-buffer() methods.
> I will try to construct an example based on your note #8 above.

Also try defining a Progress function and invoking it from a where clause.

> I good idea is to log the encountered queries from the server itself while running the set of tests. That way we should get an idea of what we should expect.

I've done this; see #1868, note 24.

Of course, we can't limit our solution to ignore client-side where clause considerations, even if they're not represented in these debug dumps. Also, consider that this output may not be correct, if we don't already handle this case with the existing solution -- I'm not sure we do. Please test this, if you manage to get such a test case to work in Progress.

#13 - 01/16/2015 02:25 PM - Ovidiu Maxiniuc

I started implementing a small 'interpreting' engine, targeting only the required node/items. In fact at this time, the interpreter is able to process the simpler FIND-FIRST / FindQuery methods.
However, when the JAST gotten more complex, in the case of OPEN QUERY / AdaptiveQuery, I am getting the following error in printed in message area:

```
    ** Logical dbname dictdb must be connected in order to add alias dictdb. (1660)
```

when I process the jast code.

The line that causes it is equivalent to:

```
    ConnectionManager.createAlias("dictdb", "p2j_test");
```

in the 'execute' method before calling the constructor fro the query.
This is odd. In the 'compiled' execution, with same JAST, this is not visible. So it is not related to a configuration/setting.

I saw that there are some recent changes regarding this alias created for the first connected database but I did not went too deep on it.

Should I ignore the node:

```
                ConnectionManager.createAlias [STATIC_METHOD_CALL] @0:0
                   dictdb [STRING] @0:0
                   p2j_test [STRING] @0:0
```

when interpreting the jast?

#14 - 01/16/2015 02:31 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> I started implementing a small 'interpreting' engine, targeting only the required node/items. In fact at this time, the interpreter is able to process the simpler FIND-FIRST / FindQuery methods.

Very cool! I am very interested to review when you have something you can share.

> However, when the JAST gotten more complex, in the case of OPEN QUERY / AdaptiveQuery, I am getting the following error in printed in message area:
>
> ```
> ** Logical dbname dictdb must be connected in order to add alias dictdb. (1660)
> ```
>
> when I process the jast code.
>
> The line that causes it is equivalent to:
>
> ```
> ConnectionManager.createAlias("dictdb", "p2j_test");
> ```
>
> in the 'execute' method before calling the constructor fro the query.

The text of the error message suggests you are passing the same parameter (dictdb) for both the alias and ldbName. Are you sure you are passing p2j_test as the ldbName in the interpreted createAlias call?

#15 - 01/16/2015 02:38 PM - Eric Faulhaber

While I am excited that you've begun implementing a solution, I'm curious whether you've found the answers to the questions we documented in notes 7-12 above. Most importantly, is it possible/legal in Progress to construct a dynamic query which will trigger client-side where clause processing?

#16 - 01/16/2015 03:28 PM - Ovidiu Maxiniuc

- File om_upd20150116a.zip added

> The text of the error message suggests you are passing the same parameter (dictdb) for both the alias and ldbName. Are you sure you are passing p2j_test as the ldbName in the interpreted createAlias call?

You are right. After using the correct node, the message disappeared.

> While I am excited that you've begun implementing a solution, I'm curious whether you've found the answers to the questions we documented in notes 7-12 above. Most importantly, is it possible/legal in Progress to construct a dynamic query which will trigger client-side where clause processing?

I did some researches on windev01. Progress is not very permissive. The static statements can be much more complex than the and dynamic calls using methods/handles.
I also downloaded and I had a look over the logs in #1868, note 24. There are no WhereExpression used in any of the queries. Instead I noticed occurrences of CompoundQuery s that will complicate the implementation.
If you have the chance to re-run it, please add the string predicate before and, to simplify the files, only log the 'final' JAST/java source as feed to im-memory compiler. The version before the post-processing does not contain really important info.

I attached an 'alpha' version. It certainly needs to be restructured.

#17 - 01/16/2015 03:30 PM - Eric Faulhaber

Related to the createAlias topic, here's an email exchange I had with Constantin recently:

> Eric,
>
> The problem here is that at conversion-time the SchemaDictionary (IIRC) needs to be aware of any explicitly created DB aliases... otherwise, when that alias is used in the query, it will not be recognized properly.
>
> Thanks,
> Constantin
>
> On 12/15/2014 1:32 AM, Eric Faulhaber wrote:
>
> > Hi Constantin,
> >
> > Do you recall why this bit of code from DynamicQueryHelper.prepareTempTable(SchemaDictionary dict, StringBuilder pcode) is necessary?

```
      ...
      // custom database aliases...
      Map<String, String> aliasesByLDB = ConnectionManager.get().getAliasesByLDB();

      for (String ldb : ldbs)
      {
```

```
            String alias = aliasesByLDB.get(ldb.toLowerCase());
            if (alias != null)
            {
               String stmt = "CREATE ALIAS %s FOR DATABASE %s.%n";

               pcode.append(String.format(stmt, alias, ldb));
            }
         }
         ...
```

Wouldn't the dynamic query code automatically pick up all aliases in the context in which it is run without this?

Thanks,
Eric

#18 - 01/16/2015 03:41 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> I did some researches on windev01. Progress is not very permissive. The static statements can be much more complex than the and dynamic calls using methods/handles.

Does that mean that neither of the two ways we discussed above of creating a client-side where clause condition are allowed in Progress? This would be very good news.

> I also downloaded and I had a look over the logs in #1868, note 24. There are no WhereExpression used in any of the queries.

Hopefully, that means there shouldn't be any, and we're doing things correctly. I guess at this point, since we're passing most of the tests and we aren't getting any errors in the server.log that suggest a client-side related, dynamic query problem, we can be pretty optimistic that this is the case.

> Instead I noticed occurrences of CompoundQuery s that will complicate the implementation.

Yes, though once we have a generic way of identifying and invoking methods and constructors, this will just represent more of the same.

> If you have the chance to re-run it, please add the string predicate before and, to simplify the files, only log the 'final' JAST/java source as feed to im-memory compiler. The version before the post-processing does not contain really important info.

Sure, I can do that, but are you certain you want to use the current, final version of the JAST as a starting point? I thought the step to get to this

version does a lot of things we no longer want to do (e.g., adding a getQuery method, adjusting instance variables, etc.). I had assumed we would be starting with the first version and modifying it quite differently to get to the final version we now need. However, I haven't looked at your alpha yet, so maybe your intent will become more clear once I do...

> I attached an 'alpha' version. It certainly needs to be restructured.

Excellent, I will have a look.

#19 - 01/16/2015 03:55 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Sure, I can do that, but are you certain you want to use the current, final version of the JAST as a starting point? I thought the step to get to this version does a lot of things we no longer want to do (e.g., adding a getQuery method, adjusting instance variables, etc.). I had assumed we would be starting with the first version and modifying it quite differently to get to the final version we now need. However, I haven't looked at your alpha yet, so maybe your intent will become more clear once I do...

My request was only to have a more insight of what are we dealing with. When possible. At this moment I prefer the post-processed JAST just because is simpler once the fix_ RULES were applied. In the final solution, this step is not needed and the access to different parts of the JAST to be done using direct method calls, building objects and calling methods by reflection, without extra processing in the JAST tree. Maybe I see this too simple, but the target is to build dynamically a P2JQuery object using JAST as a map for the pieces.

#20 - 01/16/2015 05:37 PM - Eric Faulhaber

I've taken a look at the code so far and it is a great start! Yes, I agree that the final tree is simpler to deal with for the POC. I think the approach you are taking makes a lot of sense for the alpha, however I don't want to take the implementation much further inside DynamicQueryHelper.interpret, once you're satisfied the concept works.

Much of what you are doing in the new DynamicQueryHelper.interpret method is what I think we should do in TRPL, possibly in reworked versions of convert/fix_in_mem_find_query.xml and convert/fix_in_mem_query.xml (btw, it probably makes sense to move these from rules/convert to a new rules/runtime directory, so their purpose is clear).

This is where we should be letting TRPL help us with the walk. For example, I'm thinking of something like this:

```
<variable name="parameters"   type="java.util.List" />
<variable name="methodRef"    type="java.lang.reflect.Method" />
<variable name="ctorRef"      type="java.lang.reflect.Constructor" />
<variable name="classRef"     type="java.lang.Class" />
...
<walk-rules>
   ...
   <rule>type == java.method_call or type == java.constructor
      <action>classRef = this.getAnnotation("java-class")</action>
      <!-- handle no-arg methods and c'tors immediately; there will be no descent -->
      <rule>this.numImmediateChildren == 0
```

```
         <rule>type == java.method_call
            <action>methodRef = classRef.getDeclaredMethod(text)</action>
            <!-- TODO: add error checking for null return -->
            <action>putNote("method", methodRef)</action>
         </rule>
         <rule>type == java.constructor
            <action>ctorRef = classRef.getDeclaredConstructor()</action>
            <!-- TODO: add error checking for null return -->
            <action>putNote("ctor", ctorRef)</action>
         <rule>
      </rule>
   </rule>

   <rule>
      parameters != null and
      (parent.type == java.method_call or parent.type == java.constructor)
      <action>parmTypeRef = java.getClassForAst(this)</action>
      <!-- TODO: add error checking for null return -->
      <action>parameters.add(parmTypeRef)</action>
   </rule>
   ...
</walk-rules>

<descent-rules>
   ...
   <rule>type == java.method_call or type == java.constructor
      <action>parameters = create("java.util.ArrayList")</action>
   </rule>
   ...
</descent-rules>

<ascent-rules>
   ...
   <rule>type == java.method_call or type == java.constructor
      <rule>type == java.method_call
         <action>methodRef = java.findMatchingMethod(classRef, parameters)</action>
         <!-- TODO: add error checking for null return for method -->
         <action>putNote("method", methodRef)</action>
      </rule>
      <rule>type == java.constructor
         <action>ctorRef = java.findMatchingConstructor(classRef, parameters)</action>
         <!-- TODO: add error checking for null return for constructor -->
         <action>putNote("ctor", ctorRef)</action>
      </rule>
      <action>parameters = null</action>
   </rule>
   ...
</ascent-rules>
```

As today, the source AST for this rule set is the one that comes out of the first-pass JAST processing. That step would need to be enhanced to store some annotations that we rely on here (like java-class, the java.lang.Class object that is the target of a method call or constructor call). It makes sense to do it in that earlier stage, because it is at that point that we have the most information about why we are storing a particular method or constructor in the JAST.

This rule-set would have additional rules to do the refactoring that we want (removal of all the unnecessary nodes, etc.). Or we may find that it makes more sense to do this refactoring in an intermediate rule-set that feeds into this one. In fact, we may have to, if the refactoring still needs to be different for FIND-*() and PREPARE-QUERY() JASTs.

You will note also some calls to helper functions that don't exist yet (e.g., findMatchingConstructor, findMatchingMethod, getClassForAst). These should be added to the JavaPatternWorker$JavaAstHelper inner class. It looks like you have made a good start on findMatchingMethod already.

This rule-set is very generic, and really doesn't have any domain knowledge of the methods, arguments, constructors, etc. it is walking. It's just collecting information about types, signatures, methods, constructors, and so on, and storing this in annotations which will be used in the final interpreter walk in Java code, which we want to be as easy as possible.

I know it seems like a lot of extra effort splitting the interpreter into two phases like this, but there is a method to my madness. Once this improvement is implemented, I want to be able to implement smart caching of the JASTs. To make this worthwhile, they should contain the results of as much of the expensive work as we can do up front as possible (e.g., refactoring and minimizing the JAST, collecting signatures, matching methods and constructors). The final walk to actually invoke everything via reflection should be as "thin" as possible, just doing the minimum to gather the arguments that aren't available until runtime. That way, we amortize the cost of the runtime conversion as much as possible.

#21 - 01/16/2015 05:46 PM - Eric Faulhaber

BTW, we have implemented a lot of constructor and method matching support already for TRPL itself. Perhaps we can leverage some of this previous work. See:

- com.goldencode.p2j.pattern.CommonAstSupport$Library.create
- com.goldencode.p2j.pattern.CommonAstSupport.matchConstructor
- com.goldencode.expr.SymbolResolver.matchTargetMethod
- com.goldencode.expr.SymbolResolver.resolveFunction
- com.goldencode.expr.SymbolResolver.introspectFunction

#22 - 01/17/2015 05:27 PM - Eric Faulhaber

Eric Faulhaber wrote (from Constantin's email):

> The problem here is that at conversion-time the SchemaDictionary (IIRC) needs to be aware of any explicitly created DB aliases... otherwise, when that alias is used in the query, it will not be recognized properly.

This suggests that the CREATE ALIAS statement(s) currently added before the query are there specifically for parsing (schema name resolution) purposes. If that's the case, these nodes should be removed from the Progress AST as early as possible/practical after parsing, or at least before the core conversion phase, when we generate the JAST. They should not be converted into the JAST, since they shouldn't be executed at runtime (the runtime state already includes the aliases).

#23 - 01/18/2015 08:07 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> If you have the chance to re-run it, please add the string predicate before and, to simplify the files, only log the 'final' JAST/java source as feed to im-memory compiler.

Attached to #1868, note 27.

#24 - 01/19/2015 02:36 PM - Ovidiu Maxiniuc

- File om_upd20150119a.zip added

Attached you can find a first version of 'runtime/interpret_find_query.xml'. It only handles the simple find-query cases.
To call it, the DynamicQueryHelper needs to adjusted like this:

```
                    Aast thisAst = astManager.loadTree(wa.jastFile);
```

```
                    // map buffers for quick access by their alias
                    Map<String, Buffer> buffs = new HashMap<>();
                    for (Buffer buf : locate().buffers)
                    {
                       RecordBuffer rbuf = ((BufferImpl) buf).buffer();
                       buffs.put(rbuf.getDMOAlias(), buf);
                    }
                    thisAst.putAnnotation("runtime-buffers", buffs, true);
                    ConversionPool.runTask(ConversionProfile.INTERPRET_FIND_QUERY, thisAst);// TODO for the mome
nt

                    Aast copyAst = astManager.loadTree(wa.jastFile);
                    return (P2JQuery) copyAst.getAnnotation("runtime-result", -1);
```

instead of calling the processor.postprocessJavaAst(jcode);

However, during the weekend I worked on my own approach to the runtime interpreter, in pure java. It is part of the same archive, RuntimeJastInterpreter. It has a little more functionality than the alpha POC from Friday. The idea was to have a fast runtime engine that is able to interpret pieces of the code and extract results stored in variables :

```
            qast = (JavaAst) astManager.loadTree(wa.jastFile);
            RuntimeJastInterpreter interpreter =
                  new RuntimeJastInterpreter(locate().buffers, null);
            interpreter.interpret(qast);
            return (P2JQuery) interpreter.getVariableValue("findQuery");
```

I won't continue to work on this code.

#25 - 01/19/2015 08:08 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Attached you can find a first version of 'runtime/interpret_find_query.xml'. It only handles the simple find-query cases.

This is good, closer to what I am looking for. I think you've proven the concept can work. Let's focus on a robust, generic implementation now.

One thing I'd like to refactor is to defer the invocation of methods or constructors. We don't want to do this at this stage, unless the result of the invocation is something static that will be used across multiple executions of the same query (i.e., in the not-too-distant future, when we're caching these JASTs). The TRPL work should just be about annotating information and refactoring the JAST to make it as small and fast as possible. That will allow us to figure out as little as possible on a final walk in Java code, so we can just retrieve those annotations, references to buffers, contents of

variables, and then use all this to execute.

The basic flow I'm trying to get to is:

1. JAVA: prepare-query request comes into DynamicQueryHelper; the buffers should have been added/set by this time.
2. [FUTURE JAVA: we have the buffers and predicate; it should be possible to do a cache check here; if we find an existing JAST for this combination of predicate and buffer types, skip to step 7].
3. JAVA: parse the predicate; drive TRPL runtime rules.
4. TRPL: convert up to the first pass JAST we have today, but with modifications (see below).
5. TRPL: refactor JAST to create minimalist version (only the absolute minimum nodes we need for interpret/invoke walk); we want this to have as few nodes as possible, both for speed of the final walk, and to reduce memory footprint when cached; also determine and annotate c'tors, methods, other needed info.
6. [FUTURE JAVA: cache the JAST using a cache key comprised of the buffer types (probably class name is sufficient) and original predicate].
7. JAVA: do a final walk of the JAST; extract annotations; look up the appropriate buffer when we hit a buffer reference node that we marked earlier; store the P2JQuery object when we invoke the c'tor that we marked earlier.

At that point, the rest of the code path is largely the same as it is today.

Regarding the modifications mentioned in step 4, we will need to modify the existing TRPL (convert/database_access.rules, most likely) to annotate useful information while we have it (but only in runtime conversion mode). Some examples I can think of:

- When we store a CONSTRUCTOR node for a particular query class (e.g., FindQuery), we know at this moment which query class we want to use, so we should store a "java-class" (or similar) annotation with the com.goldencode.p2j.persist.FindQuery class in that node. At this point, we don't know which variant of the c'tor we will use, but it's OK to figure that out later, in step 5.
- Also when we store this query CONSTRUCTOR node, mark it with a special annotation, so we can identify it during the final walk as the node which creates the P2JQuery we need. This is the replacement for the getQuery method we have today.
- As we add the REFERENCE node for a buffer parameter in the query c'tor or an addComponent method, mark it with a special annotation. This will tell us during the final walk that we need to retrieve the buffer which needs to be passed for this parameter. This is the replacement for today's mechanism of using reflection to store the buffers into the dynamically compiled class' public instance variables.
- When we add a CAST node to satisfy a particular method or c'tor signature (like when casting a null HQL where clause to String), we should add a "java-class" annotation for the specific class -- this will help us identify the correct method or c'tor in step 5.
- A reference to a static, final variable like LockType.NONE is essentially a constant. This currently is represented as a REFERENCE node with text LockType.NONE. We should add a "constant" annotation which stores an actual reference to a LockType.NONE object. Such an annotation can be used later, both in step 5 to determine a c'tor or method signature, as well as in step 7 as a ready-to-pass argument.

As we're implementing step 4, it is also a good time to make any improvements to the structure of the JAST which require domain-specific knowledge, to make downstream processing easier. For example, when in runtime conversion mode, we would avoid creating the STATIC_METHOD_CALL nodes for ConnectionManager.createAlias calls at this step.

Up until now, I've been thinking about step 5 being forked into different rule-sets for FIND and OPEN-QUERY features. However, I think if we do a good enough job annotating in step 4, we should be able to make this a very generic program implemented in a single XML file. Something like:

```
<rule-set>
   strip out all unneeded nodes here:
      package statement
      all import statements
      compile unit node
      class node
      all CS_* nodes
      BlockManager-related nodes
</rule-set>

<rule-set>
   add generic annotations needed by interpret/invoke walk:
      constructors
      methods
      other?
</rule-set>
```

> However, during the weekend I worked on my own approach to the runtime interpreter, in pure java. It is part of the same archive, RuntimeJastInterpreter. It has a little more functionality than the alpha POC from Friday. The idea was to have a fast runtime engine that is able to interpret pieces of the code and extract results stored in variables :
> [...]
> I won't continue to work on this code.

Don't abandon it just yet...it will not need to do as much as it does now, but this is a good candidate to morph into the "thin" walk in step 7.

#26 - 01/20/2015 03:55 PM - Ovidiu Maxiniuc

Don't abandon it just yet...it will not need to do as much as it does now, but this is a good candidate to morph into the "thin" walk in step 7.

Indeed, No matter how much object I can annotate, there will be things that cannot be processed until runtime. For example:
COMPILE_UNIT/KW_CLASS/CS_INSTANCE_METHODS/METHOD_DEF/BLOCK/ASSIGN/CONSTRUCTOR/EXPRESSION/INITIALIZER/EXPRESSION/STATIC_METHOD_CALL/CONSTRUCTOR/STRING/
corresponds to
AdaptiveQuery//new Object///isEqual/abs/decimal/1.3
that was generated from java open-query: "FOR EACH b-3 WHERE (b-3.book-id NE ", valueOf(1001), ") AND (ABS(100000000000000) eq 1.3) AND (UPPER('a') eq 'A')"
The AND -s are really constants and we should process them at conversion time.

Another idea. I think we should implement the caching soon. As you mentioned, the true constructor cannot be evaluated in TRPL, but at runtime we have the parameter types. At this moment the matching Constructor is detected and can be stored back into the JAST node as annotation. If the cache will indicate that the already built JAST can be used, the effort to detect the constructor again is eliminated. The same for methods, static methods, and even for constants or constant expressions. The first evaluation will take the "hit" but the next calls will be much faster.

#27 - 01/20/2015 04:36 PM - Eric Faulhaber

Certainly static expressions will be more efficient to convert, execute, and store at conversion. I'm pretty sure we don't do this during conversion today. If we're going to do this, we should do it for static conversion as well as runtime. I believe these expressions will convert inline in HQL currently, as long as they use only regular operators and built-in functions, which will perform reasonably well already. In any case, I think this is likely to be a larger effort with limited real-world uses (except possibly for unanticipated preprocessor output), and as such it is outside the scope of this task.

I agree that we implement the caching very soon: immediately after this task, unless we believe it will be truly trivial. If the combination of fully qualified buffer type names and query predicate string (assuming we no longer prepend a unique query name) form a reliable cache key, this should be pretty easy to implement. But again, unless the effort is just an hour or two, I don't want the scope of this task to creep.

#28 - 01/21/2015 02:58 AM - Ovidiu Maxiniuc

This is true, we don't do the constant expression evaluation at this moment neither in static conversion but it would be nice. Before the final brew, traverse the EXPRESSION tree and if constant, or parts of it, cut respective branch and replace with the computed value. However this should be deferred to another task.

My thoughts were that we should annotate more than P2JQueries. Even if my example contained toy-expression, math operations and other operators are very likely to appear in where expressions. At such a later stage (final java evaluation of even final post-processing in step 5) it is difficult (time consuming) to detect which method/constructor should be applied for abs, isEquals or decimal used in my example. We should do the runtime annotations in earlier stages (annotation/variable_definitions and convert/operators) but even at that point, the exact method is not known because the children type is not evaluated or their evaluation cannot be done at that moment. The static call is emitted, relying on java compiler to resolve the actual method to be invoked looking into the list of the ones we overload.
In consequence this final step before a node evaluation must be done after the children evaluation and because of the cost, it must be cached as annotation to the respective node for later calls and the whole TRPL tree cached as you described in steps 2 & 6.

On the other hand, if we do to many semantic annotations, the tree will grow - and the management will require additional CPU and memory, even if in the final step it will be cut, throwing away unneeded already annotated nodes. Another thing that will increase the CPU usage is that before doing the actual annotations, we should always check if we are processing in 'runtime' mode. I already created a function that does this before doing these annotations to minimize the increase of the complexity of the TRPL script.

#29 - 01/21/2015 10:19 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> The static call is emitted, relying on java compiler to resolve the actual method to be invoked looking into the list of the ones we overload.

Don't we convert these kinds of expressions into HQL today, using PL/Java UDFs to represent the built-ins being called (e.g., ABS, UPPER)?

#30 - 01/21/2015 10:27 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> On the other hand, if we do to many semantic annotations, the tree will grow - and the management will require additional CPU and memory, even if in the final step it will be cut, throwing away unneeded already annotated nodes. Another thing that will increase the CPU usage is that before doing the actual annotations, we should always check if we are processing in 'runtime' mode. I already created a function that does this before doing these annotations to minimize the increase of the complexity of the TRPL script.

Maybe a JAST is not the best data structure to store in memory for a cache (for one thing, there will be a LinkedHashMap at every annotated node, which can chew up memory quickly). Can you think of a flatter/leaner data structure which is better optimized for caching and perhaps for interpreter performance as well?

#31 - 01/21/2015 10:28 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Don't we convert these kinds of expressions into HQL today, using PL/Java UDFs to represent the built-ins being called (e.g., ABS, UPPER)?

In this case no, P2J decided to evaluate on client the expressions and pass the result as parameters:

```
query0 = new AdaptiveQuery(b3, "b3.bookId != 1001 and ? and ?", null, "b3.id asc", new Object[]
{
   isGreaterThan(abs(100000000000000L), new decimal("1.3")),
   isEqual(toUpperCase("a"), "A")
});
```

#32 - 01/22/2015 06:02 AM - Ovidiu Maxiniuc

After I got satisfactory results with simpler queries, I started to increase the complexity of the dynamic queries in order to catch new types of nodes that have to be handled/interpreted. I find in #1868 queries similar to: OPEN QUERY inMemQuery16 FOR EACH b-3 WHERE b-3.isbn <> 'Blue' AND b-3.isbn <> 'BLUE', EACH tmp WHERE tmp.af = b-3.book-id.
This is parsed to following JAST (simplified):

```
    [ASSIGN] @0:0
      query0 [REFERENCE] @0:0
      CompoundQuery [CONSTRUCTOR] @0:0
          [BOOL_TRUE] @0:0
          [BOOL_FALSE] @0:0
   addComponent [METHOD_CALL] @0:0
      query0 [REFERENCE] @0:0
      AdaptiveQuery [CONSTRUCTOR] @0:0
         b3 [REFERENCE] @0:0
         upper(b3.isbn) != 'BLUE' and upper(b3.isbn) != 'BLUE' [STRING] @0:0
          [NULL_LITERAL] @0:0
         b3.id asc [STRING] @0:0
   RecordBuffer.prepare [STATIC_METHOD_CALL] @0:0
      tmp [REFERENCE] @0:0
   addComponent [METHOD_CALL] @0:0
      query0 [REFERENCE] @0:0
      AdaptiveQuery [CONSTRUCTOR] @0:0
         tmp [REFERENCE] @0:0
         tmp.af = ? [STRING] @0:0
          [NULL_LITERAL] @0:0
         tmp.id asc [STRING] @0:0
          [EXPRESSION] @0:0
            new Object [REFERENCE_DEF] @0:0
             [INITIALIZER] @0:0
               FieldReference [CONSTRUCTOR] @0:0
                  b3 [REFERENCE] @0:0
                  bookId [STRING] @0:0
```

This would correspond to following java snippet:

```
     query0 = new CompoundQuery(true, false);
     query0.addComponent(new AdaptiveQuery(b3, "upper(b3.isbn) != 'BLUE' and upper(b3.isbn) != 'BLUE'", null,
"b3.id asc"));
     RecordBuffer.prepare(tmp);
     query0.addComponent(new AdaptiveQuery(tmp, "tmp.af = ?", null, "tmp.id asc", new Object[]
     {
        new FieldReference(b3, "bookId")
     }));
```

The trick here is that we don't have a simple object to build using a constructor, but a complex one, in which the object constructed is temporary saved and suffer some changes (addComponent()) that involves the build of other intermediary objects.

#33 - 01/22/2015 04:56 PM - Ovidiu Maxiniuc

- File om_upd20150122a.zip added

- File t2488-testcase.zip added

Uploaded update for review, together a file with my important testcases.

- I tried to detect as much as possible information in the TRPL so that the amount of reflection used in final java evaluator to be minimal. The important method that do the evaluation will check for the respective annotation and use it if found, otherwise will attempt to detect it based on available information (usually the signature formed by children). Sometimes if the exact overloaded method is not available in TRPL, at least the class name is annotated so that the java evaluator will only check the set of methods of that exact object type. In the worst case scenario, the method to be invoked is checked using the path from imports and then re-checked with strict flag turned off. In fact this is almost the same as CommonAstSupport.matchConstructor, but a little optimized.
- There is no improvement in annotating String as they evaluate the faster.
- The unneeded nodes/annotation are stripped from final JAST. In fact the evaluator does not care too much of the extra nodes because it jumps directly to the node it need. Usually the children, and when executing the method/function the statements are fast iterated using the next sibling. The only case when this happens is when an CompoundQuery is constructed.
- I know that this will a little difficult to review because of the lack of javadocs in RuntimeJastInterpreter, but I really did no had time to write them. I hope I will fix this tomorrow.

#34 - 01/23/2015 05:01 AM - Ovidiu Maxiniuc

I analyzed the logs with the queries from #1868. Here are some results/thoughts:

- there are ~5500 FIND FIRST of which ~4900 are various queries on "ttfiltercriteria", apparently on same field, "FieldName", only the value searched differs. The best long-run of queries is on FieldName = "cde": ~600, but otherwise they are rather different as keys for a cache so we would need ~2500 (estimated) cache entries.
- I am thinking of caching the already processed queries in a more intelligent way. What if we analyze the following queries:

```
    FIND FIRST ttfiltercriteria WHERE FieldName = "aBatchSubmitTreeNodeID" NO-LOCK.
    FIND FIRST ttfiltercriteria WHERE FieldName = "abatjob-num" NO-LOCK.
    FIND FIRST ttfiltercriteria WHERE FieldName = "active" NO-LOCK.
```

  and realize that they are the same ?
- We just need to be able to identify the strings / integers / other-possible parameterizable values and replace them with some kind of a token that can be later identified in the processed JAST. This way, all ~4900 occurrences would benefit from a single execution of the runtime conversion and, after the 1st evaluation (that will fill-in the missing annotations).
- Using this technique the cache will drop to size of tens or maybe few hundred which is acceptable to keep the JAST in memory.
- Implementation:
  1. parameter detection
  - string detection from the query should be simple, we just need to look after pairs of " and '
  - integer, logical detection should be rather simple, after tokenizing the query (need more thoughts on these types on implemntation, however)
  2. replace them with indexed tokens like $1, $2, etc. This way, the above queries will be grouped in same 'class':

```
FIND FIRST ttfiltercriteria WHERE FieldName = "$1" NO-LOCK.
```

  along with other 4900 similar queries.
  3. if such JAST is not already in cache, process it and save it with (parametrized-form and ttfiltercriteria) as cache key
  4. set the $1 = "aBatchSubmitTreeNodeID" (actual value) in the java interpreter before calling evaluation method
  5. when a string is encountered it is scanned for indexed tokens, if they are found, they are replaced with actual values before returning

- this works fine for for string values because the converted will consider them as string token, and it ignores their content, for other datatypes, I cannot think of a solution at this moment that would work naturally, without many changes in the conversion rules.

#35 - 01/23/2015 08:26 AM - Greg Shah

and realize that they are the same ?

Don't forget that in the Progress AST the tree structure is in fact:

- The same tree, the parameter in this case would be a STRING node. Only the text of that node would differ.
- Already properly parsed into the proper form.

I recommend detecting such cases using the Progress AST, it will be MUCH easier and MUCH more reliable. Then choose a different conversion for those nodes that must now be substitution parms. I think we may even have an annotation for that, right?

#36 - 01/23/2015 08:53 AM - Ovidiu Maxiniuc

Not sure I understand. Of course, the trees will be the except for the string value at some point. But later, during the conversion, the string will be merged back to the where expression so the node will look similar to:

```
FindQuery [CONSTRUCTOR] @0:0
    ttfiltercriteria [REFERENCE] @0:0
    "upper(ttfiltercriteria.fieldname) = 'ACTIVE'" [STRING] @0:0
    null [NULL_LITERAL] @0:0
    "ttfiltercriteria.fieldname asc" [STRING] @0:0
    LockType.NONE [REFERENCE] @0:0
```

My first idea is to avoid building any AST tree a second time. It's faster but I must admit, it's a more like heuristic approach that may fail.
I guess that the solution that combines the speed and accuracy looks like:

```
ProgressParser parser = new ProgressParser(lexer, sym);
parser.external_proc();
ProgressAst progAst = (ProgressAst) parser.getAST();
progAst.brainwash(wa.dynQueryFile, true);

// new:
List<String> params = extractParameters(progAst); // replace string (int/decimal ?) constants with $1, $2, etc
String cacheKey  = getCacheKey(progAst); // gets a unique representation of the query with constants replaced
by parameters
qast = check-cache(cacheKey, buffers)
if qast == null
      ConversionPool.runTasks();
      processor.postprocessJavaAst(jcode, false);
      store qast in cache
build interpreter
interpreter.setReplacements(params)
interpreter.evaluate()
```

The cacheKey will probably be an infix traversal of the progress tree, but any other might work.
Is this right ?

#37 - 01/23/2015 08:55 AM - Eric Faulhaber

A few thoughts on the caching:

I agree with Greg that the Progress AST will be much more reliable. However, it is another AST to store in a cache, and most likely much larger than the minimal JAST we are caching. This will more than double the memory requirement. I already was wondering whether there is a flatter/leaner data structure we could use for the JASTs. Is the JAST the most "distilled" form of information, or is there something better which could be the product of this process? Maybe even just a specialized Aast implementation which uses a leaner approach for the annotations and drops the instance variables in AnnotatedAst which we don't need. Now we would have to consider how to make the cache key much more compact as well.

I think using the buffer names (like ttfiltercriteria) as part of the key is error-prone. It happens to be done consistently in the application data we've reviewed, but I could easily write a test case which uses the same buffer name for different tables. We need to use the fully qualified name of the DMO instead, plus the predicate string. Perhaps in a follow-up, smarter implementation, we would parameterize the name of the buffer in the predicate string, so that different buffer names/aliases used for the same backing table would be recognized as the same.

Let's take a step back at this point, though. It is good to be discussing more sophisticated approaches, but I don't want to make the caching implementation particularly complicated in our first pass. It will take longer and is more likely to introduce bugs. My main goal at this point is to understand how much caching will improve performance, then we can refine the approach and perhaps give back a little of that gain to have a much smarter, leaner implementation. For now, we can achieve this early goal with a very simple implementation.

So, the approach I want to take is to start with a relatively "dumb" design, which just uses the fully qualified DMO class names and the original predicate string (without prepending any unique query names), without being overly concerned about minimizing duplicates yet. Maybe we lowercase the predicate in the key to at least eliminate duplicates caused by case differences. We should be able to implement that in just a few hours, I would think. Let's see what that buys us, and we can refine it from there.

#38 - 01/23/2015 08:58 AM - Greg Shah

When I refer to the Progress AST, I mean the original AST that is first created for the provided expression in a QUERY-PREPARE(). Don't we create a regular AST using the progress.g? That is the place to analyze and potentially mark things as substitution parms which normally would convert as inline HQL text.

I don't mean for an extra parsing step to occur. The best time to make this a parameter is during the original conversion.

Of course, we would only do this for runtime queries.

Please keep in mind that I haven't reviewed your solution, so perhaps I am misunderstanding how things work.

#39 - 01/23/2015 09:29 AM - Ovidiu Maxiniuc

Greg, you are right.

We do parse the initial query using the progress.g (ProgressParser from the code in note 36). And this is part of the current implementation. This is already compiled code so it must be fast. Yet, I have to study what the best solution for these parameters.

I will delay the exact implementation as Eric suggested.

Thank you for your valuable input.

#40 - 01/26/2015 03:14 AM - Ovidiu Maxiniuc

- File om_upd20150123a.zip added

Eric,
If you did not started the review, please find attached an update that adds the missing javadocs to RuntimeJastInterpreter.

#41 - 01/28/2015 04:21 PM - Ovidiu Maxiniuc

- File om_upd20150128b.zip added

- Status changed from New to WIP

Added 'dumb' cache implementation and some additional fixes.

#42 - 01/29/2015 02:46 PM - Ovidiu Maxiniuc

- File om_upd20150129a.zip added

This update fixes the copy/paste bug revealed by issues #2500 and #2501.

#43 - 02/05/2015 01:01 AM - Eric Faulhaber

Before I write up the code review, I need a definitive answer on the client-side where clause processing question in note 18. You wrote that Progress was not very permissive for the dynamic FINDs and that you didn't find any instances of WhereExpression in the logs. But we need to know ***for sure*** if there is any way for this to happen. If so, it will be a big hole in the current implementation.

Code review 0129a:

I really am excited about this update. Nice work overall, a great first pass. Some implementation nits to pick, nonetheless:

- I'm not crazy about adding an unused boolean parameter to Aast.putAnnotation, to keep the name but change the signature for the compiler. Please drop the boolean and just change the name to putAnnotationObject.
- Why is there a deprecated tag in a new class (JastInterpreter) for the LockType processing? This is duplicative with the code in database_access.rules, right? If it's truly dead, let's remove it. If not and it's covering some limitation in the TRPL change, can we fix it now in database_access.rules or add a TODO there?
- I think we can still do a better job of merging the FIND and OPEN QUERY fix-up TRPL into a single rule-set, but let's defer that for now. We can work incrementally toward that.
- In ConversionProfile, please remove the deprecated items. We're not moving back to the old way; we'll have to make this implementation work one way are another.
- Can the deprecated template in java_templates.tpl be removed? Again, this is a one way trip.
- Let's dump the old code in DynamicQueryHelper, we can get back to it in version control if we need to review.
- Please bump the cache size up to 2048 and put a TODO in about making its size configurable from the directory.
- Please make the debug and performance logging optional (currently going to stdout), with different levels of verbosity if necessary. We may want to output this to a separate log file.
- We don't still need to prefix a unique query name to the OPEN QUERY statements, do we?
- Why are we doing so much parsing work before checking the JAST cache (all the WorkArea updates, setting up the schema dictionary, preparing the temp-table AST, parsing the predicate)? Is this in anticipation of the smarter cache that requires a parsing pass?
- Please split the big synchronized block into two: one synchronized cache check as early in the method as possible, followed by the (unsynchronized) conversion if a cache miss occurs; then a second, synchronized block containing a cache check and cache put if the check misses again. This probably will require refactoring the try-catch-finally blocks somewhat. There may occasionally be race conditions and some wasted conversion work done, but I think the benefit of minimizing contention in this area is more important. The conversion times are very long relative to the execution times; we don't want to block other threads for tens or hundreds of milliseconds at a time.
- The JAST still seems to me to have too much class structure and method cruft. I know you key off some of these nodes in the interpreter, but I had expected you to use annotations for that instead. Not sure which is the better approach; the JAST looks bigger with the extra nodes, but in practice, dropping those nodes and using the extra annotations may actually increase the memory footprint. Let's leave it alone for now, and keep thinking about ways to make these thinner.

Please:

1. answer any questions posed above;
2. make all the changes noted above, EXCEPT:

- refining the FIND and OPEN QUERY TRPL rules;
- stripping more of the JAST nodes; and
- moving the cache check ahead of the other prep work, assuming you have a good reason for doing the cache check as late as it is.

BTW, I've run very large sets of tests using this implementation (with the larger cache size), and it's performed quite well. In the best cases, I'm seeing up to a 70% speed improvement! I've also found a few errors: some queries it can't deal with (probably would have failed with the old implementation, too), and at least one case of the JAST interpreter losing its mind on a method. I'll post these as separate Redmine issues.

#44 - 02/05/2015 03:05 PM - Ovidiu Maxiniuc

- File om_upd20150205a.zip added

Eric Faulhaber wrote:

> Before I write up the code review, I need a definitive answer on the client-side where clause processing question in note 18. You wrote that Progress was not very permissive for the dynamic FINDs and that you didn't find any instances of WhereExpression in the logs. But we need to know ***for sure*** if there is any way for this to happen. If so, it will be a big hole in the current implementation.

The dynamic FINDs are not very permissive, the problem are OPEN QUERY-es. However, there is a chance that the interpreter will handle correctly the WhereExpressions. They are in fact some variables that it will instantiate and use in the final assembly of the query. I am still searching for an example that I encounter some time ago but for the moment I have none for testing.

> Code review 0129a:
>
> - I'm not crazy about adding an unused boolean parameter to Aast.putAnnotation, to keep the name but change the signature for the compiler. Please drop the boolean and just change the name to putAnnotationObject.

Fixed. I added casts where the compiler needed guidance so the method remains overloaded.

> - Why is there a deprecated tag in a new class (JastInterpreter) for the LockType processing? This is duplicative with the code in database_access.rules, right? If it's truly dead, let's remove it. If not and it's covering some limitation in the TRPL change, can we fix it now in database_access.rules or add a TODO there?

This was a 'legacy' code from initial implementations. I removed it.

> - I think we can still do a better job of merging the FIND and OPEN QUERY fix-up TRPL into a single rule-set, but let's defer that for now. We can work incrementally toward that.

- Yes, we can. It should not be difficult to make the difference between the two cases and apply the dedicated rules.

> - In ConversionProfile, please remove the deprecated items. We're not moving back to the old way; we'll have to make this implementation work one way are another.

Done.

> - Can the deprecated template in java_templates.tpl be removed? Again, this is a one way trip.

Done.

> - Let's dump the old code in DynamicQueryHelper, we can get back to it in version control if we need to review.

Done. I only kept the BREW-ing of java code in comments for debugging purposes.

> - Please bump the cache size up to 2048 and put a TODO in about making its size configurable from the directory.

Done.

- Please make the debug and performance logging optional (currently going to stdout), with different levels of verbosity if necessary. We may want to output this to a separate log file.

Done. Currently the JAST are dumped on INFO, this might be moved to a lower level (FINER or FINEST ?).

- We don't still need to prefix a unique query name to the OPEN QUERY statements, do we?

I guess not. I hardcoded the classnames with 'DynGenQuery' so the conversion work fine. However, I kept their 'virtual' unique files because I am not fully aware if AstManager will handle that correctly.

- Why are we doing so much parsing work before checking the JAST cache (all the WorkArea updates, setting up the schema dictionary, preparing the temp-table AST, parsing the predicate)? Is this in anticipation of the smarter cache that requires a parsing pass?

- Indeed. As Greg suggested above, we should do the Progress parsing first if we need a smarter cache, so we need the lexer/parsed do their jobs and then inventing a key based on a tree with the parameters replaced by some kind of placeholders. I moved all processing to a new private method, for the moment. It made the code cleaner.

  - Please split the big synchronized block into two: one synchronized cache check as early in the method as possible, followed by the (unsynchronized) conversion if a cache miss occurs; then a second, synchronized block containing a cache check and cache put if the check misses again. This probably will require refactoring the try-catch-finally blocks somewhat. There may occasionally be race conditions and some wasted conversion work done, but I think the benefit of minimizing contention in this area is more important. The conversion times are very long relative to the execution times; we don't want to block other threads for tens or hundreds of milliseconds at a time.

Done. The race conditions are logged. They will rarely occur if ever.

- The JAST still seems to me to have too much class structure and method cruft. I know you key off some of these nodes in the interpreter, but I had expected you to use annotations for that instead. Not sure which is the better approach; the JAST looks bigger with the extra nodes, but in practice, dropping those nodes and using the extra annotations may actually increase the memory footprint. Let's leave it alone for now, and keep thinking about ways to make these thinner.

I also tried to use annotations, but this is not always possible. For example if _isLessThan is generated in one of my testcases. At conversion, we don't know which method will be used. There a 13 overloaded methods in CompareOps. Analyzing the operands at conversion time is impractical if not impossible (depending on their nature). And that in a shorter time.
Just an idea: after 1st pass of processing the JAST can be moved to a READ-ONLY state, since the last missing annotations should be already in place. At this moment, the additional nodes (like imports) can be further dropped. Also we can compute constant nodes keep their value as annotation and prune the children.

Please:

1. answer any questions posed above;
2. make all the changes noted above, EXCEPT:

- refining the FIND and OPEN QUERY TRPL rules;
- stripping more of the JAST nodes; and
- moving the cache check ahead of the other prep work, assuming you have a good reason for doing the cache check as late as it is.

I hope I answer your questions. I know that the last item was not honored but the refactoring of the code made that simpler and keeping the old order was cumbersome if not impossible.

BTW, I've run very large sets of tests using this implementation (with the larger cache size), and it's performed quite well. In the best cases, I'm seeing up to a 70% speed improvement! I've also found a few errors: some queries it can't deal with (probably would have failed with the old implementation, too), and at least one case of the JAST interpreter losing its mind on a method. I'll post these as separate Redmine issues.

I am waiting tor the new cases that fail, I hope there is one that covers the WhereExpression case I was looking for.

#45 - 02/05/2015 11:03 PM - Eric Faulhaber

Code review 0205a:

- Please put back the indent at DynamicQueryHelper:101.
- I think the logging levels used in DynamicQueryHelper are a little off. With normal use, we shouldn't see every cache action in the log. Please set logging for the cache activity (puts and gets) to FINE, as well as the elapsed time entries. Please log the race condition at INFO, and make the message text less alarming -- customers may worry unnecessarily with the current text. OTOH, the failure to interpret the JAST (line 228) represents a serious situation that requires action on our part; it should be logged as SEVERE, since that represents a broken query which likely will result in a problem at the application level. Notification that the cache is full should be at INFO level; it is a normal operation for the cache to expire elements, but it is good to know for purposes of tuning the cache size. When you obtain the logger, use LogHelper.getLogger(DynamicQueryHelper.class.getName()) instead of LogHelper.getLogger(DynamicQueryHelper.class). This will allow us to change the log level for this class in particular. The latter is scoped to the package for some reason.
- In DynamicQueryHelper.buildIndex, why did you make a change to hard-code the index annotations UNIQUE, PRIMARY, and WORD to true? That looks wrong.
- Why did you indent build.xml:265?
- What is the TODO at BufferImpl:2210 about?
- Why was DatabaseManager included in the update? It has no changes.

Otherwise, everything looks good.

Please post a new update addressing these items and run it through a conversion regression test (I don't need to review again, unless you decide to make other changes). The changes to the runtime classes aren't used in the regression test environment, so runtime regression testing won't be necessary. However, I'll run that update through a representative set of customer unit API tests to make sure it doesn't regress anything there. If that goes well, we can check it in.

#46 - 02/06/2015 07:31 AM - Ovidiu Maxiniuc

- File om_upd20150206a.zip added

- Why did you indent build.xml:265?

There were some TABs characters in that file that I replaced with plain spaces.

- What is the TODO at BufferImpl:2210 about?

The P4GL manual and method javadoc say that the ? is valid when specifying the wait mode. I tested on P4GL and this is a valid affirmation. My testcase failed to convert on P2J. We don't support this case.
BTW, for testing I used some user-defined functions that take as parameters the locking mode. I was forced to use the numerical value because P2J fails to convert these to integers, too:

```
FUNCTION execFindQuery RETURNS LOGICAL (INPUT query as CHAR, INPUT lock as INT, INPUT wait-mode as INT):
    book:FIND-FIRST(query, lock, wait-mode).
```

```
END FUNCTION.
execFindQuery("", EXCLUSIVE-LOCK, ?).                  /* fails twice in P2J conversion */
execFindQuery("where book-id eq 101", 6208, NO-WAIT).  /* converts fine */
```

Most likely the customer does not use these constructs or else the conversion would have failed, so I just added a reminder for the missing feature.

> Please post a new update addressing these items and run it through a conversion regression test (I don't need to review again, unless you decide to make other changes). The changes to the runtime classes aren't used in the regression test environment, so runtime regression testing won't be necessary. However, I'll run that update through a representative set of customer unit API tests to make sure it doesn't regress anything there. If that goes well, we can check it in.

There are no other changes in code except those requested above.
Thanks for your input for both reviews. There were issues I was not very sure about (like the log levels). The reviews clarified them for me.

#47 - 02/06/2015 12:38 PM - Eric Faulhaber

Thanks, Ovidiu. Please post your conversion results when that's done. I'll run my tests later today or overnight.

#48 - 02/06/2015 02:00 PM - Ovidiu Maxiniuc

Eric,
The last updates fail to convert the majic sources. The reason is the putAnnotation() method. Although I used cast in my code to guide the compiler to the correct variant of the overloaded method, converting the Majic code revealed other place (possible other may exist) where P2J is unable to detect the right method. The result is that conversion stops with following exception:

```
     [java] Caused by: com.goldencode.expr.AmbiguousSymbolException: com.goldencode.ast.Aast.putAnnotation
     [java]     at com.goldencode.expr.SymbolResolver.matchTargetMethod(SymbolResolver.java:1465)
     [java]     at com.goldencode.expr.SymbolResolver.introspectFunction(SymbolResolver.java:1336)
     [java]     at com.goldencode.expr.SymbolResolver.resolveFunction(SymbolResolver.java:1179)
     [java]     at com.goldencode.expr.SymbolResolver.resolveFunction(SymbolResolver.java:1120)
     [java]     at com.goldencode.expr.ExpressionParser.method(ExpressionParser.java:2042)
[...]
```

A solution is to use casts when method resolution is ambiguous for SymbolResolver but this will complicate the TRPL code too much.
The best alternative is to use your solution from note 43 and rename the new method to putAnnotationObject() so the rest of TRPL remain untouched.

#49 - 02/06/2015 02:07 PM - Eric Faulhaber

> The best alternative is to use your solution from note 43 and rename the new method to putAnnotationObject() so the rest of TRPL remain untouched.

Ok, please do this and run the test again.

#50 - 02/06/2015 04:15 PM - Ovidiu Maxiniuc

- File om_upd20150206b.zip added

The conversion test was successful. There are no differences in the generated code with the attached update.

#51 - 02/06/2015 07:03 PM - Eric Faulhaber

The regression test was successful. Please commit and distribute the update.

#52 - 02/06/2015 11:38 PM - Eric Faulhaber

- File client-where-simple.p.jast added

Each OPEN QUERY in the following test case (client-where-simple.p) will force client-side where clause processing:

```
define temp-table tt
   field f1 as integer.

function udf returns int (input num as int):
   return num.
end.

open query q1
   for each book where book.book-id > 0
        and can-find(tt where tt.f1 < book.book-id)
   no-lock.

open query q2
   for each book where udf(book.book-id) > 0
   no-lock.
```

Please try adapting these static queries into dynamic queries. I expect they will work in Progress, and we will need to figure out a way to handle them.

This test case converts to the following Java code:

```
package com.goldencode.testcases;

import com.goldencode.p2j.util.*;
import com.goldencode.p2j.persist.*;
import com.goldencode.testcases.dmo._temp.*;
import com.goldencode.testcases.dmo.p2j_test.*;
import com.goldencode.p2j.persist.lock.*;

import static com.goldencode.p2j.util.BlockManager.*;
import static com.goldencode.p2j.util.CompareOps.*;

/**
 * Business logic (converted to Java from the 4GL source code
```

```
 * in client-where-simple.p).
 */
public class ClientWhereSimple
{
   TempRecord1.Buf tt = TemporaryBuffer.define(TempRecord1.Buf.class, "tt", "tt", false);

   Book.Buf book = RecordBuffer.define(Book.Buf.class, "p2j_test", "book", "book");

   final QueryWrapper query0 = new QueryWrapper("q1", false);

   WhereExpression whereExpr0 = new WhereExpression()
   {
      public logical evaluate(final BaseDataType[] args)
      {
         return new FindQuery(tt, "tt.f1 < ?", null, (String) null, new Object[]
         {
            book.getBookId()
         }, LockType.NONE).hasOne();
      }
   };

   final QueryWrapper query1 = new QueryWrapper("q2", false);

   WhereExpression whereExpr1 = new WhereExpression()
   {
      public logical evaluate(final BaseDataType[] args)
      {
         return isGreaterThan(udf(book.getBookId()), 0);
      }
   };

   /**
    * External procedure (converted to Java from the 4GL source code
    * in client-where-simple.p).
    */
   public void execute()
   {
      externalProcedure(new Block()
      {
         public void body()
         {
            RecordBuffer.openScope(book, tt);
            RecordBuffer.prepare(tt);
            query0.assign(new AdaptiveQuery(book, "book.bookId > 0", whereExpr0, "book.bookId asc", "book-id")
);
            query0.open();
            query1.assign(new AdaptiveQuery(book, (String) null, whereExpr1, "book.bookId asc", "WHOLE-INDEX,b
ook-id", LockType.NONE));
            query1.open();
         }
      });
   }

   public integer udf(final integer _num)
   {
      return function(integer.class, new Block()
      {
         integer num = new integer(_num);

         public void body()
         {
            returnNormal(num);
         }
      });
   }
}
```

The JAST for this test case is attached. It seems like we would need to interpret the EXPRESSION subtree within the anonymous WhereExpression's subclass' evaluate method, much like we do when creating the main query. For instance, for whereExpr0:

```
<ast col="0" id="3453153706080" line="0" text="" type="EXPRESSION">
```

```
  <annotation datatype="java.lang.Long" key="peerid" value="3435973836928"/>
  <ast col="0" id="3453153706081" line="0" text="hasOne" type="METHOD_CALL">
    <annotation datatype="java.lang.Long" key="peerid" value="3435973836929"/>
    <ast col="0" id="3453153706082" line="0" text="FindQuery" type="CONSTRUCTOR">
      <annotation datatype="java.lang.Long" key="peerid" value="3435973836930"/>
      <ast col="0" id="3453153706083" line="0" text="tt" type="REFERENCE">
        <annotation datatype="java.lang.Long" key="peerid" value="3435973836931"/>
      </ast>
      <ast col="0" id="3453153706084" line="0" text="tt.f1 &lt; ?" type="STRING"/>
      <ast col="0" id="3453153706085" line="0" text="" type="NULL_LITERAL"/>
      <ast col="0" id="3453153706086" line="0" text="String" type="CAST">
        <ast col="0" id="3453153706087" line="0" text="" type="NULL_LITERAL"/>
      </ast>
      <ast col="0" id="3453153706088" line="0" text="" type="EXPRESSION">
        <ast col="0" id="3453153706089" line="0" text="new Object" type="REFERENCE_DEF">
          <annotation datatype="java.lang.Long" key="extent" value="1"/>
        </ast>
        <ast col="0" id="3453153706090" line="0" text="" type="INITIALIZER">
          <annotation datatype="java.lang.Long" key="extent" value="1"/>
          <annotation datatype="java.lang.Long" key="peerid" value="3435973836936"/>
          <ast col="0" id="3453153706091" line="0" text="" type="EXPRESSION">
            <annotation datatype="java.lang.Long" key="peerid" value="3435973836937"/>
            <ast col="0" id="3453153706092" line="0" text="getBookId" type="METHOD_CALL">
              <annotation datatype="java.lang.Long" key="peerid" value="3435973836938"/>
              <ast col="0" id="3453153706093" line="0" text="book" type="REFERENCE"/>
            </ast>
          </ast>
        </ast>
      </ast>
      <ast col="0" id="3453153706094" line="0" text="LockType.NONE" type="REFERENCE"/>
    </ast>
  </ast>
</ast>
```

The result of this evaluation would be fed into an adapter subclass of the WhereExpression abstract class.

There's more to this implementation, in that the WhereExpression class also has the getSubstitutions() method, an override for which is generated with certain client-side queries. IIRC, this would be emitted if, for instance, the can-find's where clause in the q1 query had its own query substitution parameters.

#53 - 02/09/2015 04:15 AM - Ovidiu Maxiniuc

Eric,

I tried such constructs on windev01.

CAN-FIND: P2J parses and successfully converts them in java code, but P4GL is more restrictive, even in static mode:

```
CAN-FIND is invalid within an OPEN QUERY. (3541)
**  Could not understand line 8. (196)
```

Variables: In static case, references to variables are allowed:

```
DEFINE VARIABLE kkk AS INTEGER init 100.
open query q3 FOR EACH book where book.book-id <> kkk SHARE-LOCK.
```

However, in dynamic constructs, P4GL will complain:

```
kkk must be a quoted constant or an unabbreviated, unambiguous buffer/field reference for buffers known to que
ry . (7328)
```

The only allowed solution is the one from language reference, in-lining variable's value, eventually using the QUOTER function for CHAR variables:

```
DEFINE VARIABLE my-isbn AS INTEGER init '111-111-111-111-1'.
hq:query-prepare("FOR EACH book where book.isbn <> " + quoter(my-sibn)).
```

Internal functions: are allowed. If they are applied to buffers/fields. P2J wires our UDFs in place.

User defined functions: are only allowed in static queries as in your example.
The following construct

```
hq:query-prepare("FOR EACH book where book.book-id eq udf(100)").
```

will fail at runtime with a not so explicit message similar to:

```
       ** Unable to understand after -- ".book-id eq udf". (247)
PREPARE syntax is: {FOR | PRESELECT} EACH  OF.. WHERE ... etc". (7324)
```

However, I also think that is good to support this kind of JAST trees. Otherwise we could get uncovered by some exotic case we could not think of for the moment.

#54 - 02/09/2015 08:25 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> The result of this evaluation would be fed into an adapter subclass of the WhereExpression abstract class.
> There's more to this implementation, in that the WhereExpression class also has the getSubstitutions() method, an override for which is generated with certain client-side queries. IIRC, this would be emitted if, for instance, the can-find's where clause in the q1 query had its own query substitution parameters.

That's the idea I'm working with. The problem is that the evaluate() method can be called multiple times, and each time, the result must be a new object. We cannot store that value in the adapter. The adapter must be dynamic, to rebuild the result from the 'environment' at the time of the respective call. This will happen at a different time that the interpretation of the 'main' execute and building of the query. So a reference to the interpreter is kept in the adapter. I don't think that this event will collide with our cache, otherwise some complex synchronization will have to be implemented.

Another issue at this moment, if we decide to support UDFs (P4GL doesn't, see my previous note-53), how will they be accessed. It could go naturally is the class that declare it is imported statically.

#55 - 02/09/2015 09:24 AM - Ovidiu Maxiniuc

Regarding the user-defined functions. I was thinking too ahead. At this moment the ProgressParser cannot handle UDF-s because they are not 'visible' in the parsed string ("WHERE udf(book-id) eq udf(100)"):

```
line 1:23: unexpected token: udf
    at com.goldencode.p2j.uast.ProgressParser.lvalue(ProgressParser.java:12680)
    [...]
    at com.goldencode.p2j.uast.ProgressParser.statement(ProgressParser.java:5712)
    at com.goldencode.p2j.uast.ProgressParser.single_block(ProgressParser.java:4568)
    at com.goldencode.p2j.uast.ProgressParser.block(ProgressParser.java:4327)
    at com.goldencode.p2j.uast.ProgressParser.external_proc(ProgressParser.java:4253)
    at com.goldencode.p2j.persist.DynamicQueryHelper.generateJavaTree(DynamicQueryHelper.java:328)
    at com.goldencode.p2j.persist.DynamicQueryHelper.parse(DynamicQueryHelper.java:180)
    at com.goldencode.p2j.persist.DynamicQueryHelper.parseFindQuery(DynamicQueryHelper.java:674)
    [...]
```

I guess this is the case for P4GL too, because of the compile error when a bad function name is encountered in a static open query is similar to the runtime warning issued when encountering any UDF in dynamic mode.

#56 - 02/09/2015 10:36 AM - Eric Faulhaber

If we cannot come up with a single test case which would result in client-side where clause processing (all so far seem disallowed in the Progress environment), I hesitate to spend any more time working on this feature. How would we test the implementation (if it's even necessary)?

#57 - 02/09/2015 12:17 PM - Eric Faulhaber

Stanislav, I don't fully understand the following comment, from DynamicQueryHelper:763:

```
// Dynamic buffers have unique DMO aliases, so "some-buffer" name may be converted to the
// alias like "someTable__1", while aliases in predicate will be converted using standard
// P2J rules, which will result to the converted alias "someTable". So below we explicitly
// specify converted DMO alias like "someTable__1" for the conversion rules using
// "force_dmo_alias" annotation.
```

You are the original author of this comment AFAICT. Maybe you can explain it to me. I'm seeing dynamic queries which are otherwise identical, except for the DMO alias suffix, like __1, __2, etc. We have implemented a cache for dynamic queries, based in part on the buffer name. There are cases where the buffer type and predicate of separate queries are identical, but the buffer names differ by this suffix only. It seems like these are basically the same queries in Progress, and this differentiating suffix is applied by P2J. As a result, the queries are each deemed to be unique by the caching logic, so each time we encounter one, a cache miss occurs. Thus, we add many copies of what is otherwise the same query to the cache.

Any insight into the purpose of this differentiation would be appreciated.

#58 - 02/09/2015 01:08 PM - Stanislav Lomany

Eric, are you basically asking why suffixes are added to the names of dynamic buffers?

#59 - 02/09/2015 01:13 PM - Eric Faulhaber

Stanislav Lomany wrote:

> Eric, are you basically asking why suffixes are added to the names of dynamic buffers?

Yes, why and where/how? I want to figure out if this is still relevant/necessary with the new implementation.

#60 - 02/09/2015 01:25 PM - Stanislav Lomany

IIRC the point is that there can be multiple dynamic buffers with the same name in the same scope and some P2J structures use buffer name as a key.

#61 - 02/09/2015 02:16 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> If we cannot come up with a single test case which would result in client-side where clause processing (all so far seem disallowed in the Progress environment), I hesitate to spend any more time working on this feature. How would we test the implementation (if it's even necessary)?

I finally have found such cases (thanks to Constantin for help):

```
DEFINE TEMP-TABLE tt
   FIELD f1 AS INTEGER.

hQuery:SET-BUFFERS(BUFFER book:HANDLE, BUFFER tt:HANDLE).

hQuery:QUERY-PREPARE("FOR EACH book, FIRST tt WHERE BUFFER tt:NAME = book.isbn").

hQuery:QUERY-PREPARE("FOR EACH book, EACH tt WHERE BUFFER tt::f1 = book.book-id").
```

These are probably not the single cases, but they are the proof that we have to handle the WhereExpression in interpreter. I am still looking for a test case that need overriding the getSubstitutions() method.

Some things look strange on P4GL using : and :: punctuation:

- the equality test in the where clause is never yes (for example: WHERE tt.f2 = buffer book:name, even if tt.f2 is a char initialized with "book")
- WHERE BUFFER tt::f1 <> book.book-id will result in a ** Incompatible data types in expression or assignment. (223)

#62 - 02/10/2015 01:52 PM - Ovidiu Maxiniuc

Eric,
I almost finished implementing support for 'extending' and interpreting the client-side WhereExpression but I this moment (among other issues) I encountered the following error in the dynamic conversion:
Consider the query to prepare: "FOR EACH book, EACH tt WHERE BUFFER tt:NAME = book.isbn".
The converted JAST dumped to console looks like:

```
OPEN QUERY DynGenQuery FOR EACH book, EACH tt WHERE BUFFER tt:NAME = book.isbn
compilation unit [COMPILE_UNIT] @0:0
```

```
com.goldencode.p2j.util.* [KW_IMPORT] @0:0
com.goldencode.p2j.util.BlockManager.* [STATIC_IMPORT] @0:0
DynGenQuery [KW_CLASS] @0:0
    [CS_INSTANCE_VARS] @0:0
      = [ASSIGN] @0:0
         WhereExpression [REFERENCE_DEF] @0:0
         WhereExpression [ANON_CTOR] @0:0
             [CS_INSTANCE_METHODS] @0:0
               evaluate [METHOD_DEF] @0:0
                   [LPARENS] @0:0
                     BaseDataType[] [REFERENCE_DEF] @0:0
                  block [BLOCK] @0:0
                      [KW_RETURN] @0:0
                        isEqual [STATIC_METHOD_CALL] @0:0
                           tt [REFERENCE] @0:0
                           getIsbn [METHOD_CALL] @0:0
                              book [REFERENCE] @0:0
      CompoundQuery [REFERENCE_DEF] @0:0
    [CS_INSTANCE_METHODS] @0:0
      execute [METHOD_DEF] @0:0
      [...]
```

Well, at the moment of execution of the query, the evaluate is called in my WhereExpressionAdapter. But, if you look closely you'll notice that the BUFFER tt:NAME is converted to tt [REFERENCE] @0:0 instead of name [METHOD_CALL]/tt [REFERENCE]. The Other operand of the isEqual method seems to be processed correctly: getIsbn [METHOD_CALL]/book [REFERENCE].

Is it possible that the simplified runtime conversion is missing processing step? Is this something related to temp-tables maybe ?

#63 - 02/10/2015 01:57 PM - Eric Faulhaber

At a glance, yes, that looks wrong. Have you converted the static form of the same query? This will tell you if the problem is related to runtime conversion specifically, as opposed to a general conversion problem.

#64 - 02/10/2015 02:07 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> At a glance, yes, that looks wrong. Have you converted the static form of the same query? This will tell you if the problem is related to runtime conversion specifically, as opposed to a general conversion problem.

Yes, the statically converted query looks fine:

```
WhereExpression whereExpr0 = new WhereExpression()
{
   public logical evaluate(final BaseDataType[] args)
   {
      return isEqual(tt.name(), book.getIsbn());
   }
};
```

#65 - 02/10/2015 02:41 PM - Ovidiu Maxiniuc

- File om_upd20150210a.zip added

The attached update should handle the WhereExpressions.
There are a few other issues that must be solved first:

- the correct dynamic generation of the JAST-s (see my note-62 above).
- the openQueryProcessor.preparePredicate(String). At this moment the method invalidates all queries that contain COLON character.
- the correct generation of particular cases of dereference ::. The where expression for
  "FOR EACH book, EACH tt WHERE BUFFER tt::f1 <> book.book-id")
  is generated as:

```
WhereExpression whereExpr1 = new WhereExpression() {
   public logical evaluate(final BaseDataType[] args) {
      return isNotEqual(((handle) tt).unwrapDereferenceable().dereference("f1"), book.getBookId());
   }
}
```

  This is wrong, the cast (handle) tt is incorrect.
- I notice some bugs in P4GL that we are not support. I will add another task for this.

#66 - 02/11/2015 06:56 AM - Ovidiu Maxiniuc

The problem reported in note-62 is fixed by adding the methods_attributes.rules to runtime conversion. With this occasion I removed the brewing rules that are not used any more.

I discovered yet another case of bad conversion. Constructs like

```
OPEN QUERY q1 FOR EACH book, EACH tt WHERE book.book-id < SESSION:TIMEZONE AND NOT buffer tt:AVAILABLE.
```

are wrongly transformed (at both conversion and run time) to:

```
WhereExpression whereExpr0 = new WhereExpression() {
```

```
      public logical evaluate(final BaseDataType[] args) {
         return isLessThan(book.getBookId(), SessionUtils.getSessionTimeZone());
      }
   };
   query2.addComponent(new AdaptiveQuery(tt, (String) null, whereExpr0, "tt.id asc", new Object[] {
               and(isLessThan(book.getBookId(), SessionUtils.getSessionTimeZone()), new LogicalExpression() {
                  public logical execute() {
                     return not(tt.available());
                  }
               })
            }));
```

#67 - 02/16/2015 10:21 PM - Eric Faulhaber

Code review 0210a:

The version of DynamicQueryHelper in the update shows no difference from the latest version in bzr. Was this intentional? The rest looks good to me.

#68 - 02/17/2015 03:33 PM - Ovidiu Maxiniuc

Changes since 0210a:

- Improved preparePredicate() for OPEN QUERY predicates to
  - allow one level nested quotes
  - allow handle methods as valid constructs: buffer book:name
  - allow dereference operator (:: punctuation): buffer book::isbn
  - drop garbage after the COLON (:) or DOT (.) when they are terminators (followed by some white space).
  See #2401.
- Added support for dynamic calling of methods from super classes
- Fixed dereference operator when applied directly to Buffer.
- Prevented a NPE when listeners is empty in ChangeBroker.
- Added support for method and dereference in dynamic conversions.

#69 - 02/17/2015 03:34 PM - Ovidiu Maxiniuc

- File om_upd20150217a.zip added

Oops, I forgot the update. here it is.

#70 - 02/18/2015 02:40 PM - Ovidiu Maxiniuc

- File om_upd20150218a.zip added

Added support for LogicalExpression anon c'tor.
Refactored some method names to better fit their implementation.

#71 - 02/18/2015 03:40 PM - Eric Faulhaber

Code review 0218a:

Where did the need to improve the predicate prep come from? What is the rationale behind the cutoff point at which this level of sanity checking should end and the rest be left up to the Progress parser?

If you remove the brew-related rules from the "runtime-rules" pattern set in build.xml, isn't that going to disable some of the debug output capability in DynamicQueryHelper? For example, I don't run the P2J server from within eclipse; I use the jar files. This change will disable that debug capability for me.

In core_conversion.xml, please consolidate your header entries into one. There generally should be one header entry per check-in of a file.

Everything else looks good to me.

If the above are the only changes you make, please go ahead with a conversion regression test (since some of the changed files are used in static conversion) and commit the update when it passes. The change to ChangeBroker is in a method that should never be invoked by the regression test environment, and the changes to the uast classes are additive only, so runtime testing is not necessary.

#72 - 02/19/2015 09:44 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Where did the need to improve the predicate prep come from? What is the rationale behind the cutoff point at which this level of sanity checking should end and the rest be left up to the Progress parser?

Sorry, I should have give more details in note-68 above. The old implementation of predicate preparation for OPEN QUERY had some flaws: there were two of most importance:

- it was parsing wrongly the string literals (a literal started with first of ' or " and ended at second ' or " so a mix of quotes was validated), leading into truncation of queries like: "for each book where isbn = 'abc~". abc'" to "for each book where isbn = 'abc~""
- it wasn't allowing the ":" and "::" punctuation at all. Predicates like "for each book where timezone(book.timestamp) eq session:timezone" were incorrectly reported as being syntactically incorrect.
  Normally the ., :, and :: operators are followed (without any blank in between) by a field / method / dereference. At the moment of this scan, we cannot tell if the right-hand of the punctuation is correct, we need the true ProgressParser but it will act at a later stage. However, if a blank character (one of \r\n\t) occurs after the . or : Progress just stop parsing the string, ignoring the rest of the characters.

> If you remove the brew-related rules from the "runtime-rules" pattern set in build.xml, isn't that going to disable some of the debug output capability in DynamicQueryHelper? For example, I don't run the P2J server from within eclipse; I use the jar files. This change will disable that debug capability for me.

This is true. But the JASTs are still available. At least that's all I use. This is the reason also for the small additions in Aast / DumpTree. I think it's a good tool for debugging: to optionally see the annotations for AST nodes. Nevertheless, seeing the true java code is important, too, so I will add those files back to jar.

> If the above are the only changes you make, please go ahead with a conversion regression test (since some of the changed files are used in static conversion) and commit the update when it passes. The change to ChangeBroker is in a method that should never be invoked by the regression test environment, and the changes to the uast classes are additive only, so runtime testing is not necessary.

OK.

#73 - 02/19/2015 12:01 PM - Ovidiu Maxiniuc

- File om_upd20150219a.zip added

The generated code is unchanged for majic. The attached update passed the conversion test.

#74 - 02/19/2015 03:04 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> The generated code is unchanged for majic. The attached update passed the conversion test.

Looks good. Please commit and distribute.

#75 - 02/26/2015 12:49 PM - Ovidiu Maxiniuc

I found a test case that fails to convert properly.

```
DEF TEMP-TABLE tt0
   FIELD f1 AS INT.
DEFINE VARIABLE t3 AS HANDLE.
CREATE BUFFER t3 FOR TABLE "tt0" BUFFER-NAME "t3h".
t3:FIND-FIRST().
```

At runtime the last line generate the following in-memory dynamic code:

```
DEFINE BUFFER t3h FOR TEMP-TABLE tt0.
FIND FIRST t3h  SHARE-LOCK.
```

It will then be processed to JAST that, before applying post-processing step, look like this (annotation included):

```
compilation unit [COMPILE_UNIT] @0:0
   com.goldencode.p2j.persist.dynquery [KW_PACKAGE] @0:0
   com.goldencode.p2j.util.* [KW_IMPORT] @0:0
   com.goldencode.p2j.util.BlockManager.* [STATIC_IMPORT] @0:0
   DynGenQuery [KW_CLASS] @0:0
   (access=public, pkgname=com.goldencode.p2j.persist.dynquery)
       [CS_CONSTANTS] @0:0
       [CS_STATIC_VARS] @0:0
       [CS_STATIC_INITS] @0:0
       [CS_INSTANCE_VARS] @0:0
         = [ASSIGN] @0:0
            TempRecord1.Buf [REFERENCE_DEF] @0:0
            (name=t3h)
            TemporaryBuffer.define [STATIC_METHOD_CALL] @0:0
               TempRecord1.Buf.class [REFERENCE] @0:0
               t3h [STRING] @0:0
               tt0 [STRING] @0:0
                [BOOL_FALSE] @0:0
         = [ASSIGN] @0:0
            TempRecord1.Buf [REFERENCE_DEF] @0:0
            (name=t3hBuf2)
            TemporaryBuffer.define [STATIC_METHOD_CALL] @0:0
               t3h [REFERENCE] @0:0
```

```
               t3hBuf2 [STRING] @0:0
               t3h [STRING] @0:0
       [CS_CONSTRUCTORS] @0:0
       [CS_STATIC_METHODS] @0:0
       [CS_INSTANCE_METHODS] @0:0
         execute [METHOD_DEF] @0:0
         (peerid=25769803777)
             [BLOCK] @0:0
               externalProcedure [STATIC_METHOD_CALL] @0:0
                  Block [ANON_CTOR] @0:0
                      [CS_INSTANCE_VARS] @0:0
                      [CS_INSTANCE_METHODS] @0:0
                        body [METHOD_DEF] @0:0
                        (access=public, rettype=void)
                           block [BLOCK] @0:0
                           (peerid=25769803777)
                              buffer_anchor [BOGUS] @0:0
                                 RecordBuffer.openScope [STATIC_METHOD_CALL] @0:0
                                 (peerid=25769803807)
                                    t3hBuf2 [REFERENCE] @0:0
                                    t3h [REFERENCE] @0:0
                              first [METHOD_CALL] @0:0
                              (peerid=25769803791)
                                 FindQuery [CONSTRUCTOR] @0:0
                                 (peerid=25769803797, runtime-class=class com.goldencode.p2j.persist.FindQuery
)
                                    t3hBuf2 [REFERENCE] @0:0
                                    (peerid=25769803798)
                                    String [CAST] @0:0
                                        [NULL_LITERAL] @0:0
                                     [NULL_LITERAL] @0:0
                                    t3hBuf2.id asc [STRING] @0:0
                                    LockType.SHARE [REFERENCE] @0:0
                                    (runtime-value=SHARE)
       [CS_INNER_CLASSES] @0:0
   com.goldencode.p2j.persist.* [KW_IMPORT] @0:0
   com.goldencode.testcases.dmo._temp.* [KW_IMPORT] @0:0
   com.goldencode.p2j.persist.lock.* [KW_IMPORT] @0:0
```

Note that there are two TempRecord1.Buf defined: t3h and t3hBuf2. Unfortunately, the FindQuery statement will use the second one, the original t3h remains unused. The problem is that, when interpreting this JAST, the t3hBuf2 is unknown, the caller only supplies informations about the original t3 / t3h buffer: t3h = com.goldencode.p2j.persist.$__Proxy1@727e748c. Of course, the dynamic query fails with following messages:

```
(com.goldencode.p2j.persist.DynamicQueryHelper:SEVERE) Failed to interpret JAST for QueryCacheKey{query='', bu
fferNames=[t3h], bufferDMOs=[class com.goldencode.testcases.dmo._temp.impl.TempRecord1Impl]}

com.goldencode.p2j.persist.RuntimeJastInterpreter$InterpreterException: No REFERENCE named (t3hBuf2)

(ErrorManager:SEVERE) {00000001:00000008:bogus} Invalid WHERE clause in FIND method for buffer tt1. (10041)
```

#76 - 03/06/2015 01:31 PM - Ovidiu Maxiniuc

- File om_upd20150306a.zip added

The strings literals are generated as ready-to-be-written to java source files and the escape sequences must be processed to obtain their correct value.

#77 - 03/06/2015 05:56 PM - Eric Faulhaber

Code review 0306a:

Looks good. I'll run tests with this update and let you know how it goes.

#78 - 03/18/2015 01:23 PM - Ovidiu Maxiniuc

- File om_upd20150318a.zip added

The update adds DateExpression and IntegerExpression implementation. They might be needed in dynamic queries.

#79 - 03/24/2015 10:24 AM - Eric Faulhaber

Eric Faulhaber wrote:

> Code review 0306a:
>
> Looks good. I'll run tests with this update and let you know how it goes.

Sorry, forgot to get back to you. Testing went well with this update, please commit and distribute.

#80 - 03/24/2015 10:28 AM - Eric Faulhaber

Code review 0318a:

Looks good, noticed one typo on line 1121: "...some king of expression..."

What testing have you done with this update applied?

#81 - 03/24/2015 10:44 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Code review 0318a:
>
> Looks good, noticed one typo on line 1121: "...some king of expression..."

Thanks!

> What testing have you done with this update applied?

See #2535, my note 10. There are IntegerExpression and DateExpression.
The integer variant occurs from first note (at first I thought that was the issue in #2535). But then I reproduced the bug in static mode. With the update applied, the IntegerExpression is not generated any more for that testcase. But, to be sure we don't get held in the future, I think it's better to add support for int and date expressions, at least.

#82 - 03/24/2015 11:03 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> What testing have you done with this update applied?

> See #2535, my note 10. There are IntegerExpression and DateExpression.
> The integer variant occurs from first note (at first I thought that was the issue in #2535). But then I reproduced the bug in static mode. With the update applied, the IntegerExpression is not generated any more for that testcase. But, to be sure we don't get held in the future, I think it's better to add support for int and date expressions, at least.

Go ahead and commit and distribute this update, but we also should add support for all the data wrapper types in a follow-up update.

#83 - 03/27/2015 01:21 PM - Ovidiu Maxiniuc

The update om_upd20150318a was committed to bzr as revno 10824 and distributed by mail.

#84 - 04/16/2015 10:43 AM - Eric Faulhaber

Is there any work left to do on the initial implementation, or any known problems outstanding? If not, I will close this task, and we can handle any new bugs as separate issues.

#85 - 04/16/2015 10:57 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Is there any work left to do on the initial implementation, or any known problems outstanding? If not, I will close this task, and we can handle any new bugs as separate issues.

The RuntimeJastInterpreter still lacks the support for some client-side where __Expression-s (currently only Logical, Integer and Date were detected as required and supported). We don't know if the other datatypes are really needed.

Some improvements in the caching can be implemented. The current 'dumb' solution will duplicate the efforts for queries like "FIND FIRST book WHERE isbn EQ 'isbn1'" and "FIND FIRST book WHERE isbn EQ 'isbn2'". They are the same in fact, they only differ by a string literal that can be

detected and extracted as a SQL parameter.

#86 - 05/12/2015 11:47 AM - Ovidiu Maxiniuc

- File om_upd20150512a.zip added

The attached update allows variables to be declared and initialized in the dynamically constructed object. They were stripped when the JAST was postprocessed.
The issue was discovered for sorted queries, when the addSortCriterion methods use byExprN member fields that was only declared and not properly initialized any more.

#87 - 05/12/2015 12:37 PM - Eric Faulhaber

Code review 0512a:

The change looks good. If it fixes your problem locally, please commit and distribute, since regression testing won't cover this functionality.

#88 - 05/12/2015 01:53 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Code review 0512a:
>
> The change looks good. If it fixes your problem locally, please commit and distribute, since regression testing won't cover this functionality.

Committed revision 10859. Distributed my mail.

#89 - 06/08/2015 02:22 PM - Eric Faulhaber

Please investigate the following error from the P2J server.log:

```
com.goldencode.p2j.persist.RuntimeJastInterpreter$InterpreterException: Failed to execute static method
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.callStaticMethod(RuntimeJastInterpreter.java:844)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.evalExpression(RuntimeJastInterpreter.java:965)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.execMethod(RuntimeJastInterpreter.java:388)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.access$000(RuntimeJastInterpreter.java:73)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter$WhereExpressionAdapter.evaluate(RuntimeJastInterp
reter.java:1457)
        at com.goldencode.p2j.persist.WhereExpression.evaluate(WhereExpression.java:206)
        at com.goldencode.p2j.persist.RandomAccessQuery.execute(RandomAccessQuery.java:2578)
        at com.goldencode.p2j.persist.RandomAccessQuery.first(RandomAccessQuery.java:1167)
        at com.goldencode.p2j.persist.RandomAccessQuery.first(RandomAccessQuery.java:1113)
        at com.goldencode.p2j.persist.CompoundQuery.processComponent(CompoundQuery.java:2228)
        at com.goldencode.p2j.persist.CompoundQuery.retrieveImpl(CompoundQuery.java:1975)
        at com.goldencode.p2j.persist.CompoundQuery.retrieve(CompoundQuery.java:1552)
        at com.goldencode.p2j.persist.CompoundQuery.retrieve(CompoundQuery.java:1438)
        at com.goldencode.p2j.persist.CompoundQuery.next(CompoundQuery.java:877)
        at com.goldencode.p2j.persist.CompoundQuery.next(CompoundQuery.java:823)
        at com.goldencode.p2j.persist.AbstractQuery.wrapNext(AbstractQuery.java:2199)
        at com.goldencode.p2j.persist.AbstractQuery.getNext(AbstractQuery.java:1242)
        at com.goldencode.p2j.persist.QueryWrapper.getNext(QueryWrapper.java:3668)
        ...
```

See server.log from #1868, note 60, or any other recent server.log from a run of the full set of search unit tests.

#90 - 06/09/2015 03:19 PM - Ovidiu Maxiniuc

I was able to duplicate the issue. One occurrence is in test[Entity=|+TNCY_TENANT| TestType=|SEARCH| TestSubType=|SEARCHEABLE_ATTRIBUTE| xmlSuffix=|26|].
There is an incorrect JAST tree for a LogicalExpression implementation. The execute looks like this:

```
execute [METHOD_DEF] @0:0 (access=public, rettype=logical)
   block [BLOCK] @0:0
       [KW_RETURN] @0:0
         isLessThan [STATIC_METHOD_CALL] @0:0
            getTermDat [METHOD_CALL] @0:0(runtime-class=class com.goldencode.p2j.persist.$__Proxy72, runtime-m
ethod=public com.goldencode.p2j.util.date com.goldencode.p2j.persist.$__Proxy72.getTermDat())
               tncy [REFERENCE] @0:0
```

As you can see, the isLessThan has only one child node, so the evaluator will fail to find such a static method in the declared places.
BBS with more details.

#91 - 06/09/2015 03:52 PM - Ovidiu Maxiniuc

The P4GL converted query is:

```
OPEN QUERY DynGenQuery  FOR EACH prhst , FIRST tncy WHERE tncy.prhst-occ-num = prhst.occ-num AND ((tncy.term-d
at = ? and prhst.end-dte < TODAY) OR tncy.term-dat < TODAY) NO-LOCK
```

It seems that the generated code is bad (before applying the dynamic post-processing). The java class generated looks like this:

```
package com.goldencode.p2j.persist.dynquery;

import com.goldencode.p2j.util.*;
import com.goldencode.p2j.persist.*;
import com.something.server.dmo.<db_name>.*;
import com.goldencode.p2j.persist.lock.*;

import static com.goldencode.p2j.util.BlockManager.*;
import static com.goldencode.p2j.util.CompareOps.*;
import static com.goldencode.p2j.util.logical.*;

public class DynGenQuery
{
   Tncy.Buf tncy = RecordBuffer.define(Tncy.Buf.class, "<db_name>", "tncy", "tncy");

   Prhst.Buf prhst = RecordBuffer.define(Prhst.Buf.class, "<db_name>", "prhst", "prhst");

   WhereExpression whereExpr0 = new WhereExpression()
   {
      public Object[] getSubstitutions()
      {
         return new Object[]
         {
            new DateExpression()
            {
                              public date execute()
```

```
                {
                   return date.today();
                }
             }
          };
       }

       public logical evaluate(final BaseDataType[] args)
       {
          return or(and(isUnknown(tncy.getTermDat()), new LogicalExpression()
          {
             public logical execute()
             {
*                return isLessThan(prhst.getEndDte());
             }
          }), new LogicalExpression()
          {
             public logical execute()
             {
*                return isLessThan(tncy.getTermDat());
             }
          });
       }
    };

    public void execute()
    {
       externalProcedure(new Block()
       {
          CompoundQuery query0 = null;

          public void body()
          {
             RecordBuffer.openScope(tncy, prhst);
             query0 = new CompoundQuery(true, false);
             query0.addComponent(new AdaptiveQuery(prhst, (String) null, null, "prhst.prSeqNo asc, prhst.seqNo
asc, prhst.perNum asc", "WHOLE-INDEX,idx-1"));
             RecordBuffer.prepare(tncy);
             query0.addComponent(new RandomAccessQuery(tncy, "tncy.prhstOccNum = ?", whereExpr0, "tncy.prhstOcc
Num asc", "occ-num", new Object[]
             {
                new FieldReference(prhst, "occNum")
             }, LockType.NONE), QueryConstants.FIRST);
             query0.open();
          }
       });
    }
}
```

As you can see at first sight, there are two places (I inserted an * at the beginning of the line) that are not syntactically correct.
Looking closer, we can see that the second TODAY member for both < operator (extracted only once in the substitution), is missing in the evaluate return expression.

I will create a static testcase and go further from there. I don't think that this issue is related to this task.

LE: GES removed the customer's name and customer's database name from this entry.

#92 - 06/10/2015 01:55 PM - Ovidiu Maxiniuc

- File om_upd20150610a.zip added

I have a fix for the issue. There are 3 important changes:

1. the convert/variable_references.rules was added to runtime query conversion. This was the 'visible' part, that caused the invalid code to be generated at runtime (the substitution arguments within a client-side where clause expression were not processed in dynamic mode), now, the generated code is fine.
   (**Note:** *there are a number of fixes in the later months that require new rules files to be added to runtime bundle. The size of them is not a problem, but processing them at runtime could add a performance hit because of the extra tests (even if nothing is changes in the AST trees, the rather consuming functions are evaluated). I wonder if these required pieces of TRPL code could be extracted in separate files somehow?*)
2. after the code was correctly generated, the interpreter failed to process it because the particular code was accessing parameters of a method. I added a ScopedDictionary to keep the actual values of the parameters. As in Java, they are passed by value. Constructs like this.OuterClass.someVar are not supported yet. As in Java, in the case of collision, the parameters are preferred instead of declared variables. The variables are still stored as a plain Map instead of a ScopedDictionary, too.
3. support for arrays was added. Both indexed and array reference are supported.

The same testcase provided an example of a cast for date objects. I added support for all major datatypes from P2J.

#93 - 06/11/2015 01:46 AM - Eric Faulhaber

Code review 0610a:

Looks good. Assuming your local testing confirms it fixes the problem, please run conversion regression testing only (the changes to the conversion/build are very minor, but just to be safe...).

Note there are some commented out settings added (used while testing, perhaps?) in the aspectj target of build.xml. Please remove, then commit/distribute when testing passes.

> (**Note:** *there are a number of fixes in the later months that require new rules files to be added to runtime bundle. The size of them is not a problem, but processing them at runtime could add a performance hit because of the extra tests (even if nothing is changes in the AST trees, the rather consuming functions are evaluated). I wonder if these required pieces of TRPL code could be extracted in separate files somehow?*)

Can you identify specific areas/tests/functions that concern you? Have you profiled/measured to confirm your concerns, or is it more of a gut feeling based on reviewing the code? TRPL code in general is pretty fast once it has been compiled, though if most of it is unnecessary, clearly it will be faster to avoid running that code at all.

#94 - 06/11/2015 12:28 PM - Eric Faulhaber

I tested the 0610a update with the full set of search unit tests. I can confirm that it fixed the problematic test and did not regress others.

#95 - 06/11/2015 01:48 PM - Ovidiu Maxiniuc

- File om_upd20150611a.zip added

The attached update is basically the same as 0610a except for the comment form build.xml removed and a cascading if structure replaced with the equivalent switch statement. It passed the conversion testing with Majic.

#96 - 06/11/2015 01:58 PM - Eric Faulhaber

Please commit and distribute 0611a. Thanks.

#97 - 09/04/2015 12:39 AM - Eric Faulhaber

Ovidiu, I need some help extending RuntimeJastInterpreter to deal with a new way of handling query substitution parameters and to support a new Java token type: LAMBDA.

In order to fix some outstanding bugs which require the deferred evaluation of some query substitution parameters, I implemented the conversion of those query substitution parameters as lambda expressions. A typical case would look like this:

```
new FindQuery(foo, "foo.bar = ?", ..., new Object[]
{
   (P2JQuery.Parameter) () -> toUpperCase(bar.getFoo())
}, LockType.NONE).first();
```

Another might be:

```
new FindQuery(foo, "foo.bar = ?", null, "foo.bar asc", new Object[]
{
   (P2JQuery.Parameter) () -> userDefinedFunction(bar.getFoo())
}, LockType.NONE).first();
```

When such a query is dynamically prepared, it hits the RuntimeJastInterpreter looking something like this:

```
...
FindQuery [CONSTRUCTOR] @0:0
   foo [REFERENCE] @0:0
   foo.bar = ?) [STRING] @0:0
    [NULL_LITERAL] @0:0
   foo.bar asc [STRING] @0:0
   idx-1 [STRING] @0:0
    [EXPRESSION] @0:0
      new Object [REFERENCE_DEF] @0:0
       [INITIALIZER] @0:0
         P2JQuery.Parameter [CAST] @0:0
             [LAMBDA] @0:0
                [LPARENS] @0:0
               character.toUpperCase [STATIC_METHOD_CALL] @0:0
                  getFoo [METHOD_CALL] @0:0
                     bar [REFERENCE] @0:0
   LockType.NONE [REFERENCE] @0:0
QueryConstants.FIRST [REFERENCE] @0:0
...
```

This isn't necessarily perfectly accurate, but the idea is to convey the structure of the new form of query substitution parameter. Of course, RJI doesn't know what to do with the LAMBDA token, so we end up with:

```
com.goldencode.p2j.persist.RuntimeJastInterpreter$InterpreterException: Unknown expression type
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.evalExpression(RuntimeJastInterpreter.java:1135)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.evalExpression(RuntimeJastInterpreter.java:936)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.evalExpression(RuntimeJastInterpreter.java:996)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.evalExpression(RuntimeJastInterpreter.java:951)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.collectParameters(RuntimeJastInterpreter.java:622
```

```
)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.callCtor(RuntimeJastInterpreter.java:552)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.evalExpression(RuntimeJastInterpreter.java:924)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.collectParameters(RuntimeJastInterpreter.java:622
)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.callMethod(RuntimeJastInterpreter.java:702)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.execMethod(RuntimeJastInterpreter.java:433)
        at com.goldencode.p2j.persist.RuntimeJastInterpreter.interpret(RuntimeJastInterpreter.java:230)
        at com.goldencode.p2j.persist.DynamicQueryHelper.parse(DynamicQueryHelper.java:241)
        at com.goldencode.p2j.persist.DynamicQueryHelper.parseQuery(DynamicQueryHelper.java:602)
        at com.goldencode.p2j.persist.QueryWrapper.prepare(QueryWrapper.java:3437)
        ...
```

The key here is that the lambda expression should not be evaluated by the interpreter, it just needs to be cast to the right functional interface (hence the cast to P2JQuery.Parameter), and passed into the query c'tor in the Object array. So, hopefully, the changes are pretty straightforward.

The updates are in branch 2650a/10936, which is based on trunk 10930. Could you please take a look and add this support? When you have a fix, please check it into that branch.

You will need to build P2J with Java 8, so you'll have to make that your default environment (java and javac) temporarily. Don't worry about getting the native code working -- it will build, but it may not run appserver properly. That isn't necessary for this work and it caused problems for me, so please don't spend any time on it. I've gotten around the issues, but I haven't documented how yet. Thanks.

#98 - 01/30/2016 04:06 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Eric Faulhaber wrote:
>
> > Is there any work left to do on the initial implementation, or any known problems outstanding? If not, I will close this task, and we can handle any new bugs as separate issues.
>
> The RuntimeJastInterpreter still lacks the support for some client-side where __Expression-s (currently only Logical, Integer and Date were detected as required and supported). We don't know if the other datatypes are really needed.

How much effort do you estimate is necessary to implement the remaining datatypes?

> Some improvements in the caching can be implemented. The current 'dumb' solution will duplicate the efforts for queries like "FIND FIRST book WHERE isbn EQ 'isbn1'" and "FIND FIRST book WHERE isbn EQ 'isbn2'". They are the same in fact, they only differ by a string literal that can be detected and extracted as a SQL parameter.

This can be moved into a separate issue as an idea for a performance or efficiency enhancement. It is not a functional requirement.

#99 - 02/01/2016 09:10 AM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> Ovidiu Maxiniuc wrote:
>
> > The RuntimeJastInterpreter still lacks the support for some client-side where __Expression-s (currently only Logical, Integer and Date were detected as required and supported). We don't know if the other datatypes are really needed.
>
> How much effort do you estimate is necessary to implement the remaining datatypes?

This is a matter of minutes, to duplicate classes/methods and replacing the new datatypes.

> > Some improvements in the caching can be implemented. The current 'dumb' solution will duplicate the efforts for queries like "FIND FIRST book WHERE isbn EQ 'isbn1'" and "FIND FIRST book WHERE isbn EQ 'isbn2'". They are the same in fact, they only differ by a string literal that can be detected and extracted as a SQL parameter.
>
> This can be moved into a separate issue as an idea for a performance or efficiency enhancement. It is not a functional requirement.

OK, I will create a new task to track this as a feature/enhancement.

#100 - 02/01/2016 09:34 AM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Eric Faulhaber wrote:
>
> > Ovidiu Maxiniuc wrote:
> >
> > > The RuntimeJastInterpreter still lacks the support for some client-side where __Expression-s (currently only Logical, Integer and Date were detected as required and supported). We don't know if the other datatypes are really needed.

> How much effort do you estimate is necessary to implement the remaining datatypes?

> This is a matter of minutes, to duplicate classes/methods and replacing the new datatypes.

OK, please add this to your next task branch (after 2975a). Better we spend the minutes now than potentially hours tracking down a bug from this later. Then we can close this task.

#101 - 02/01/2016 12:48 PM - Ovidiu Maxiniuc

Support for interpreting remaining datatypes __Expressions in RuntimeJastInterpreter was added as rev. 10969 in task branch 2975a.

#102 - 02/01/2016 02:27 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> Support for interpreting remaining datatypes __Expressions in RuntimeJastInterpreter was added as rev. 10969 in task branch 2975a.

Did you find that task branch 2975a causes converted code changes in Majic or the customer server project, such that we now need to do runtime regression testing? I would have expected not, since otherwise we should have seen compile failures in those projects before the 2975a/10968 fix.

If there were no conversion differences in Majic or customer server project, please back out the __Expression changes (i.e., rev 10969) from 2975a and place those changes instead into the next task branch you create for working on a different issue, after #2975. Don't uncommit the change, but please put the old versions of the runtime classes back as they were and/or remove any new classes, then commit that as 2975a/10970 (should be the same as 2975a/10968).

2975a should only contain the conversion changes to fix the regression covered by #2975. Otherwise, we have to do runtime regression testing for 2975a. Also, I don't want to put this in a dedicated task branch, which would cause a dedicated round of regression testing just for this change.

Thanks.

#103 - 02/01/2016 02:45 PM - Ovidiu Maxiniuc

Eric, I reverted 2975a back to rev 10968 = 10970. Sorry, I misread your note 100.

#104 - 02/02/2016 10:56 AM - Ovidiu Maxiniuc

The updated was committed to task branch 2794a as revision 10970.

#105 - 02/02/2016 12:12 PM - Eric Faulhaber

Ovidiu Maxiniuc wrote:

> The updated was committed to task branch 2794a as revision 10970.

The changes are good, thanks.

I think we'll want to parameterize the expression class hierarchy at some point, since there's a lot of duplication (unless there's some functional or performance concern in doing so). The base classes (and conversion) originally were written before Java had generics. I know this wasn't feasible here, since the classes you inherit from are emitted in converted code, but we'll want to change that with a future code improvement.

#106 - 02/02/2016 12:30 PM - Ovidiu Maxiniuc

Eric Faulhaber wrote:

> I think we'll want to parameterize the expression class hierarchy at some point, since there's a lot of duplication (unless there's some functional or performance concern in doing so). The base classes (and conversion) originally were written before Java had generics. I know this wasn't feasible here, since the classes you inherit from are emitted in converted code, but we'll want to change that with a future code improvement.

This is true, I also thought about doing it now, but there are some architectural issues here. All the classes involved share the same ancestor (Resolvable) but the class hierarchy split too early into specialized classes. Once the generics are added to base/intermediary classes we will replace all these utilities with a generic one.

#107 - 03/25/2016 01:26 PM - Eric Faulhaber

- % Done changed from 0 to 100

- Status changed from WIP to Closed

The purpose of this issue has been achieved, so I am closing this task. If we decide to make a change to the caching implementation, it will be tracked under #2275.

#108 - 11/16/2016 12:06 PM - Greg Shah

- Target version changed from Milestone 11 to Cleanup and Stablization for Server Features

Files

ecf_upd20150115a.zip	14.1 KB	01/15/2015	Eric Faulhaber

om_upd20150116a.zip	17.6 KB	01/16/2015	Ovidiu Maxiniuc
om_upd20150119a.zip	46.2 KB	01/19/2015	Ovidiu Maxiniuc
om_upd20150122a.zip	185 KB	01/22/2015	Ovidiu Maxiniuc
t2488-testcase.zip	1.58 KB	01/22/2015	Ovidiu Maxiniuc
om_upd20150123a.zip	188 KB	01/26/2015	Ovidiu Maxiniuc
om_upd20150128b.zip	200 KB	01/28/2015	Ovidiu Maxiniuc
om_upd20150129a.zip	201 KB	01/29/2015	Ovidiu Maxiniuc
om_upd20150205a.zip	235 KB	02/05/2015	Ovidiu Maxiniuc
om_upd20150206a.zip	210 KB	02/06/2015	Ovidiu Maxiniuc
om_upd20150206b.zip	210 KB	02/06/2015	Ovidiu Maxiniuc
client-where-simple.p.jast	20.4 KB	02/07/2015	Eric Faulhaber
om_upd20150210a.zip	67.2 KB	02/10/2015	Ovidiu Maxiniuc
om_upd20150217a.zip	91.6 KB	02/17/2015	Ovidiu Maxiniuc
om_upd20150218a.zip	92.2 KB	02/18/2015	Ovidiu Maxiniuc
om_upd20150219a.zip	92.2 KB	02/19/2015	Ovidiu Maxiniuc
om_upd20150306a.zip	12.4 KB	03/06/2015	Ovidiu Maxiniuc
om_upd20150318a.zip	12.7 KB	03/18/2015	Ovidiu Maxiniuc
om_upd20150512a.zip	2.23 KB	05/12/2015	Ovidiu Maxiniuc
om_upd20150610a.zip	26.9 KB	06/10/2015	Ovidiu Maxiniuc
om_upd20150611a.zip	26.9 KB	06/11/2015	Ovidiu Maxiniuc