ARQ - Frequently Asked Questions

java.lang.NoClassDefFoundError

> The classpath is wrong. Include all the jar files in lib/. You need to name each one. See also "The CLASSPATH and Environment Variables"

java.lang.NoClassDefFoundError : Exception in thread "main"

> The classpath is wrong. Include all the jar files in lib/ before running one of the command line applications.

java.lang.NoSuchFieldError: actualValueType

> This is almost always due to using the wrong version of the Xerces library. Jena and ARQ make use of XML schema support that changed at Xerces 2.6.0 and is not compatible with earlier versions. At the time of writing Jena ships with Xerces 2.6.1.
>
> In some situations your runtime environment may be picking up an earlier version of Xerces from an "endorsed" directory. You will need to either disable use of that endorsed library or replace it by a more up to date version of Xerces. This appears to happen with some distributions of Tomcat 5.* and certain configurations of JDK 1.4.1.

Query Debugging

> Look at the data in N3 or Turtle or N-triples. This can give you a better sense of the graph than RDF/XML.
>
> Use the command line tools and a sample of your data to develop a query, especially a complex one.
>
> Break your query up into smaller sections.

How do I do test substrings of literals?

> SPARQL provides regular expression matching which can be used to test for substrings and other forms that SQL's LIKE operator provides.
>
> Example: find resource with an RDFS label contains the substring "orange", matching without respecting case of the string.

```
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?x
WHERE {
  ?x rdfs:label ?v .
  FILTER regex(?v, "orange", "i")
}
```

> The regular expression matching in ARQ is provided by java.util.regex.

Accented characters and characters outside of basic latin ~ SPARQL queries are assumed to be Unicode strings. If typing from a text editor, ensure it is working in UTF-8 and not the operating system native

character set. UTF-8 is not the default character set under MS Windows.

```
ARQ supports \\u escape sequences in queries for the input of 16bit
codepoints.

ARQ does not support 32 bit codepoints (it would require a move to
Java 1.5, including all support libraries and checking the codebase
for char/codepoint inconsistencies and drop support for Java 1.4).

The same is true for data. XML files can be written in any
XML-supported character set if the right `?xml` processing
instruction is used. The default is UTF-8 or UTF-16.
```

XSD DateTime

Examples of correctly formatted XSD DateTime literals are: these two are actually the same point in time and will test equal in a filter:

```
"2005-04-04T04:04:04Z"^^xsd:dateTime
"2004-12-31T18:01:00-05:00"^^<http://www.w3.org/2001/XMLSchema#dateTime>
```

- The timezone is required.
- The datatype must be given.

String Operations

ARQ provides many of the XPath/XQuery functions and operators including string operations. These include: `fn:contains`, `fn:starts-with`, `fn:ends-with`. See the library page for details of all function provided.

Note 1: For string operations taken from XQuery/XPath, character positions are numbered from 1, unlike Java where they are numbered from 0.

Note 2: `fn:substring` operation takes the length of the substring as the 3rd argument, unlike Java where it is the end index.

Tutorial - Manipulating SPARQL using ARQ

When you've been working with SPARQL you quickly find that static queries are restrictive. Maybe you want to vary a value, perhaps add a filter, alter the limit, etc etc. Being an impatient sort you dive in to the query string, and it works. But what about little Bobby Tables? And, even if you sanitise your inputs, string manipulation is a fraught process and syntax errors await you. Although it might seem harder than string munging, the ARQ API is your friend in the long run.

Originally published on the Research Revealed project blog

Inserting values (simple prepared statements)

Let's begin with something simple. Suppose we wanted to restrict the following query to a particular person:

```
select * { ?person <http://xmlns.com/foaf/0.1/name> ?name }
```

`String#replaceAll` would work, but there is a safer way. `QueryExecutionFactory` in most cases lets you supply a `QuerySolution` with which you can prebind values.

```
QuerySolutionMap initialBinding = new QuerySolutionMap();
initialBinding.add("name", personResource);
qe = QueryExecutionFactory.create(query, dataset, initialBinding);
```

This is often much simpler than the string equivalent since you don't have to escape quotes in literals. (Beware that this doesn't work for `sparqlService`, which is a great shame. It would be nice to spend some time remedying that.)

Making a Query from Scratch

The previously mentioned limitation is due to the fact that prebinding doesn't actually change the query at all, but the execution of that query. So what how do we really alter queries?

ARQ provides two ways to work with queries: at the syntax level (`Query` and `Element`), or the algebra level (`Op`). The distinction is clear in filters:

```
SELECT ?s { ?s <http://example.com/val> ?val . FILTER ( ?val < 20 ) }
```

If you work at the syntax level you'll find that this looks (in pseudo code) like:

```
(GROUP (PATTERN ( ?s <http://example.com/val> ?val )) (FILTER ( < ?val 20 ) ))
```

That is there's a group containing a triple pattern and a filter, just as you see in the query. The algebra is different, and we can see it using `arq.qparse --print op`

```
$ java arq.qparse --print op 'SELECT ?s { ?s <http://example.com/val> ?val .
FILTER ( ?val < 20 ) }'
(base <file:///...>
    (project (?s)
        (filter (< ?val 20)
            (bgp (triple ?s <http://example.com/val> ?val)))))
```

Here the filter contains the pattern, rather than sitting next to it. This form makes it clear that the expression is filtering the pattern.

Let's create that query from scratch using ARQ. We begin with some common pieces: the triple to match, and the expression for the filter.

```
// ?s ?p ?o .
Triple pattern =
    Triple.create(Var.alloc("s"), Var.alloc("p"), Var.alloc("o"));
// ( ?s < 20 )
Expr e = new E_LessThan(new ExprVar("s"), new NodeValueInteger(20));
```

`Triple` should be familiar from jena. `Var` is an extension of `Node` for variables. `Expr` is the root interface for expressions, those things that appear in `FILTER` and `LET`.

First the syntax route:

```
ElementTriplesBlock block = new ElementTriplesBlock(); // Make a BGP
block.addTriple(pattern);                              // Add our pattern match
ElementFilter filter = new ElementFilter(e);           // Make a filter matching
the expression
ElementGroup body = new ElementGroup();                // Group our pattern
```

```
match and filter
   body.addElement(block);
   body.addElement(filter);

   Query q = QueryFactory.make();
   q.setQueryPattern(body);                                  // Set the body of the
query to our group
   q.setQuerySelectType();                                   // Make it a select query
   q.addResultVar("s");                                      // Select ?s
```

Now the algebra:

```
   Op op;
   BasicPattern pat = new BasicPattern();                    // Make a pattern
   pat.add(pattern);                                         // Add our pattern match
   op = new OpBGP(pat);                                      // Make a BGP from this
pattern
   op = OpFilter.filter(e, op);                              // Filter that pattern
with our expression
   op = new OpProject(op, Arrays.asList(Var.alloc("s")));    // Reduce to just ?s
   Query q = OpAsQuery.asQuery(op);                          // Convert to a query
   q.setQuerySelectType();                                   // Make is a select query
```

Notice that the query form (`SELECT, CONSTRUCT, DESCRIBE, ASK`) isn't part of the algebra, and we have to set this in the query (although SELECT is the default). `FROM` and `FROM NAMED` are similarly absent.

Navigating and Tinkering: Visitors

You can also look around the algebra and syntax using visitors. Start by extending `OpVisitorBase` (`ElementVisitorBase`) which stubs out the interface so you can concentrate on the parts of interest, then walk using `OpWalker.walk(Op, OpVisitor)` (`ElementWalker.walk(Element, ElementVisitor)`). These work bottom up.

For some alterations, like manipulating triple matches in place, visitors will do the trick. They provide a simple way to get to the right parts of the query, and you can alter the pattern backing BGPs in both the algebra and syntax. Mutation isn't consistently available, however, so don't depend on it.

Transforming the Algebra

So far there is no obvious advantage in using the algebra. The real power is visible in transformers, which allow you to reorganise an algebra completely. ARQ makes extensive use of transformations to simplify and optimise query execution.

In Research Revealed I wrote some code to take a number of constraints and produce a query. There were a number of ways to do this, but one way I found was to generate ops from each constraint and join the results:

```
   for (Constraint con: cons) {
       op = OpJoin.create(op, consToOp(cons)); // join
   }
```

The result was a perfectly correct mess, which is only barely readable with just three conditions:

```
   (join
```

```
      (join
          (filter (< ?o0 20) (bgp (triple ?s <urn:ex:prop0> ?o0)))
          (filter (< ?o1 20) (bgp (triple ?s <urn:ex:prop1> ?o1))))
      (filter (< ?o2 20) (bgp (triple ?s <urn:ex:prop2> ?o2))))
```

Each of the constraints is a filter on a bgp. This can be made much more readable by moving the filters out, and merging the triple patterns. We can do this with the following `Transform`:

```
   class QueryCleaner extends TransformBase
   {
       @Override
       public Op transform(OpJoin join, Op left, Op right) {
           // Bail if not of the right form
           if (!(left instanceof OpFilter && right instanceof OpFilter)) return
join;
           OpFilter leftF = (OpFilter) left;
           OpFilter rightF = (OpFilter) right;

           // Add all of the triple matches to the LHS BGP
           ((OpBGP) leftF.getSubOp()).getPattern().addAll(((OpBGP)
rightF.getSubOp()).getPattern());
           // Add the RHS filter to the LHS
           leftF.getExprs().addAll(rightF.getExprs());
           return leftF;
       }
   }
   ...
   op = Transformer.transform(new QueryCleaner(), op); // clean query
```

This looks for joins of the form:

```
   (join
       (filter (exp1) (bgp1))
       (filter (exp2) (bgp2)))
```

And replaces it with:

```
   (filter (exp1 && exp2) (bgp1 && bgp2))
```

As we go through the original query all joins are removed, and the result is:

```
   (filter (exprlist (< ?o0 20) (< ?o1 20) (< ?o2 20))
       (bgp
           (triple ?s <urn:ex:prop0> ?o0)
           (triple ?s <urn:ex:prop1> ?o1)
           (triple ?s <urn:ex:prop2> ?o2)
   ))
```

That completes this brief introduction. There is much more to ARQ, of course, but hopefully you now have a taste for what it can do.

LARQ - adding free text searches to SPARQL

LARQ is a combination of ARQ and Lucene. It gives users the ability to perform free text searches within their SPARQL queries. Lucene indexes are additional information for accessing the RDF graph, not storage for the graph itself.

Some example code is available here: https://svn.apache.org/repos/asf/jena/trunk/jena-larq/src/test/java/org/apache/jena/larq/examples/.

Two helper commands are provided: `larq.larqbuilder` and `larq.larq` used respectively for updating and querying LARQ indexes.

A full description of the free text query language syntax is given in the Lucene query syntax document.

Usage Patterns

There are three basic usage patterns supported:

- Pattern 1 : index string literals. The index will return the literals matching the Lucene search pattern.
- Pattern 2 : index subject resources by string literal. The index returns the subjects with property value matching a text query.
- Pattern 3 : index graph nodes based on strings not present in the graph.

Patterns 1 and 2 have the indexed content in the graph. Both 1 and 2 can be modified by specifying a property so that only values of a given property are indexed. Pattern 2 is less flexible as discussed below. Pattern 3 is covered in the external content section below.

LARQ can be used in other ways as well but the classes for these patterns are supplied. In both patterns 1 and 2, strings are indexed, being plain strings, string with any language tag or any literal with datatype XSD string.

Index Creation

There are many ways to use Lucene, which can be set up to handle particular features or languages. The creation of the index is done outside of the ARQ query system proper and only accessed at query time. LARQ includes some platform classes and also utility classes to create indexes on string literals for the use cases above. Indexing can be performed as the graph is read in, or to built from an existing graph.

Index Builders

An index builder is a class to create a Lucene index from RDF data.

- `IndexBuilderString`: This is the most commonly used index builder. It indexes plain literals (with or without language tags) and XSD strings and stores the complete literal. Optionally, a property can be supplied which restricts indexing to strings in statements using that property.
- `IndexBuilderSubject`: Index the subject resource by a string literal, an store the subject resource, possibly restricted by a specified property.

Lucene has many ways to create indexes and the index builder classes do not attempt to provide all possible Lucene features. Applications may need to extend or modify the standard index builders provided by LARQ.

Index Creation

An index can be built while reading RDF into a model:

```
// -- Read and index all literal strings.
IndexBuilderString larqBuilder = new IndexBuilderString() ;

// -- Index statements as they are added to the model.
model.register(larqBuilder) ;

FileManager.get().readModel(model, datafile) ;

// -- Finish indexing
larqBuilder.closeWriter() ;
model.unregister(larqBuilder) ;

// -- Create the access index
IndexLARQ index = larqBuilder.getIndex() ;
```

or it can be created from an existing model:

```
// -- Create an index based on existing statements
larqBuilder.indexStatements(model.listStatements()) ;
// -- Finish indexing
larqBuilder.closeWriter() ;
// -- Create the access index
IndexLARQ index = larqBuilder.getIndex() ;
```

Index Registration

Next the index is made available to ARQ. This can be done globally:

```
// -- Make globally available
LARQ.setDefaultIndex(index) ;
```

or it can be set on a per-query execution basis.

```
QueryExecution qExec = QueryExecutionFactory.create(query, model) ;
// -- Make available to this query execution only
LARQ.setDefaultIndex(qExec.getContext(), index) ;
```

In both these cases, the default index is set, which is the one expected by property function `pf:textMatch`. Use of multiple indexes in the same query can be achieved by introducing new properties. The application can subclass the search class `org.apache.jena.larq.LuceneSearch` to set different indexes with different property names.

Query using a Lucene index

Query execution is as usual using the property function pf:textMatch. "textMatch" can be thought of as an implied relationship in the data. Note the prefix ends in ":".

```
String queryString = StringUtils.join("\n", new String[]{
        "PREFIX pf: <http://jena.hpl.hp.com/ARQ/property#>",
        "SELECT * {" ,
        "    ?lit pf:textMatch '+text'",
        "}"
    }) ;
Query query = QueryFactory.create(queryString) ;
QueryExecution qExec = QueryExecutionFactory.create(query, model) ;
```

```
ResultSetFormatter.out(System.out, qExec.execSelect(), query) ;
```

The subjects with a property value of the matched literals can be retrieved by looking up the literals in the model:

```
PREFIX pf: <http://jena.hpl.hp.com/ARQ/property#>
SELECT ?doc
{
    ?lit pf:textMatch '+text' .
    ?doc ?p ?lit
}
```

This is a more flexible way of achieving the effect of using a `IndexBuilderSubject`. `IndexBuilderSubject` can be more compact when there are many large literals (it stores the subject not the literal) but does not work for blank node subjects without extremely careful co-ordination with a persistent model. Looking the literal up in the model does not have this complication.

Accessing the Lucene Score

The application can get access to the Lucene match score by using a list argument for the subject of `pf:textMatch`. The list must have two arguments, both unbound variables at the time of the query.

```
PREFIX pf: <http://jena.hpl.hp.com/ARQ/property#>
SELECT ?doc ?score
{
    (?lit ?score ) pf:textMatch '+text' .
    ?doc ?p ?lit
}
```

Limiting the number of matches

When used with just a query string, pf:textMatch returns all the Lucene matches. In many applications, the application is only interested in the first few matches (Lucene returns matches in order, highest scoring first), or only matches above some score threshold. The query argument that forms the object of the pf:textMatch property can also be a list, including a score threshold and a total limit on the number of results matched.

```
?lit pf:textMatch ( '+text' 100 ) .        # Limit to at most 100 hits

?lit pf:textMatch ( '+text' 0.5 ) .        # Limit to Lucene scores of 0.5 and
over.

?lit pf:textMatch ( '+text' 0.5 100 ) .    # Limit to scores of 0.5 and limit to
100 hits
```

Direct Application Use

The IndexLARQ class provides the ability to search programmatically, not just from ARQ. The searchModelByIndex method returns an iterator over RDFNodes.

```
// -- Create the access index
IndexLARQ index = larqBuilder.getIndex() ;
```

```
NodeIterator nIter = index.searchModelByIndex("+text") ;
for ( ; nIter.hasNext() ; )
{
    // if it's an index storing literals ...
    Literal lit = (Literal)nIter.nextNode() ;
}
```

External Content

- Pattern 3: index graph nodes based on strings not present in the graph.

Sometimes, the index needs to be created based on external material and the index gives nodes in the graph. This can be done by using `IndexBuilderNode` which is a helper class to relate external material to some RDF node.

Here, the indexed content is not in the RDF graph at all. For example, the indexed content may come from HTML.XHTML, PDFs or XML documents and the RDF graph only holds the metadata about these content items.

The Lucene contributions page lists some content converters.

Getting Help and Getting Involved

If you have a problem with LARQ, make sure you read the Getting help with Jena page and post a message on the users@jena.apache.org mailing list. You can also search the jena-users mailing list archives here.

If you use LARQ and you want to get involved, make sure you read the Getting Involved page. You can help us making LARQ better by:

- improving this documentation, writing tutorials or blog posts about LARQ
- letting us know how you use LARQ, your use cases and what are in your opinion missing features
- answering users question about LARQ on the users@jena.apache.org mailing list
- submitting bug reports and feature requests on JIRA: https://issues.apache.org/jira/browse/JENA
- voting or submitting patches for the currently open bugs or improvements for LARQ
- checking out LARQ source code, playing with it and let us know your ideas for possible improvements: https://svn.apache.org/repos/asf/jena/trunk/jena-larq

ARQ - RDF Collections

RDF collections, also called RDF lists, are difficult to query directly.

ARQ provides a 3 property functions to work with RDF collections.

- `list:member` -- members of a list
- `list:index` -- index of a member in a list
- `list:length` -- length of a list

`list:member` is similar to `rdfs:member` except for RDF lists. ARQ also provides

`rdfs:member`.

See the property functions library page.

ARQ - Writing Filter Functions

Applications can add SPARQL functions to the query engine. This is done by writing a class implementing the right interface, then either registering it or using the fake `java:` URI scheme to dynamically call the function.

Writing SPARQL Value Functions

A SPARQL value function is an extension point of the SPARQL query language that allows URI to name a function in the query processor.

In the ARQ engine, code to implement function must implement the interface `com.hp.hpl.jena.sparql.function.Function` although it is easier to work with one of the abstract classes for specific numbers of arguments like `com.hp.hpl.jena.sparql.function.FunctionBase1` for one argument functions. Functions do not have to have a fixed number of arguments.

The abstract class `FunctionBase`, the superclass of `FunctionBase1` to `FunctionBase4`, evaluates its arguments and calls the implementation code with argument values (if a variable was unbound, an error will have been generated)

It is possible to get unevaluated arguments but care must be taken not to violate the rules of function evaluation. The standard functions that access unevaluated arguments are the logical 'or' and logical 'and' operations that back `||` and && are special forms to allow for the special exception handling rules.

Normally, function should be a pure evaluation based on it's argument. It should not access a graph nor return different values for the same arguments (to allow expression optimization). Usually, these requirements can be better met with a property function. Functions can't bind a variables; this would be done in a property function as well.

Example: (this is the max function in the standard ARQ library):

```
public class max extends FunctionBase2
{
    public max() { super() ; }
    public NodeValue exec(NodeValue nv1, NodeValue nv2)
    {
        return Functions.max(nv1, nv2) ;
    }
}
```

The function takes two arguments and returns a single value. The class NodeValue represents values and supports value-based operations. NodeValue value support includes the XSD datatypes, xsd:decimal and all it's subtypes like xsd:integer and xsd:byte, xsd';double, xsd:float, xsd:boolean, xsd:dateTime and xsd:date. Literals with language tags are also treated as values in additional "value spaces" determined by the language tag without regard to case.

The `Functions` class contains the core XML Functions and Operators operations. Class NodeFunctions contains the implementations of node-centric operations like `isLiteral` and `str`.

If any of the arguments are wrong, then the function should throw `ExprEvalException`.

Example: calculate the canonical namespace from a URI (calls the Jena operation for the actual work):

```
public class namespace extends FunctionBase1
{
    public namespace() { super() ; }

    public NodeValue exec(NodeValue v)
    {
        Node n = v.asNode() ;
        if ( ! n.isURI() )
            throw new ExprEvalException("Not a URI: "+FmtUtils.stringForNode(n)) ;
        String str = n.getNameSpace() ;
        return NodeValue.makeString(str) ;
    }
}
```

This throws an evaluation exception if it is passed a value that is not a URI.

The standard library, in package `com.hp.hpl.jena.sparql.function.library`, contains many examples.

Registering Functions

The query compiler finds functions based on the functions URI. There is a global registry of known functions, but any query execution can have it's own function registry.

For each function, there is a function factory associated with the URI. A new function instance is created for each use of a function in each query execution.

```
// Register with the global registry.
FunctionRegistry.get().put("http://example.org/function#myFunction", new
MyFunctionFactory()) ;
```

A common case is registering a specific class for a function implementation so there is an addition method that takes a class, wraps in a built-in function factory and registers the function implementation.

```
// Register with the global registry.
FunctionRegistry.get().put("http://example.org/function#myFunction",
MyFunction.class) ;
```

Another convenience route to function calling is to use the `java:` URI scheme. This dynamically loads the code, which must be on the Java classpath. With this scheme, the function URI gives the class name. There is automatic registration of a wrapper into the function registry. This way, no explicit registration step is needed by the application and queries issues with the command line tools can load custom functions.

```
PREFIX f: <java:app.myFunctions.>
...
   FILTER f:myTest(?x, ?y)
...
   FILTER (?x + f:myIntToXSD(?y))
...
```

ARQ - Building Queries Programmatically

It is possible to build queries by building and abstract syntax tree (as the parser does) or by building the algebra expression for the query. It is usually better to work with the algebra form as it is more regular.

See the examples in the ARQ `src-examples/` directory of the ARQ distribution, particularly `arq.examples.AlgebraExec`.

See also ARQ - SPARQL Algebra

RQ - SPARQL Algebra

A SPARQL query in ARQ goes through several stages of processing:

- String to Query (parsing)
- Translation from Query to a SPARQL algebra expression
- Optimization of the algebra expression
- Query plan determination and low-level optimization
- Evaluation of the query plan

This page describes how to access and use expressions in the SPARQL algebra within ARQ. The definition of the SPARQL algebra is to be found in the SPARQL specification in section 12. ARQ can be extended to modify the evaluation of the algebra form to access difefrent graph storage implementations.

The classes for the datastructures for the algebra resize in the package `com.hp.hpl.jena.sparql.algebra` in the `op` subpackage. All the classes are names "`Op...`"; the interface that they all offer is "`Op`".

Viewing the algebra expression for a Query

The command line tool arq.qparse will print the algebra form of a query:

```
arq.qparse --print=op --query=Q.rq
arq.qparse --print=op 'SELECT * { ?s ?p ?o}'
```

The syntax of the output is SSE, a simple format for writing data structures involving RDF terms. It can be read back in again to produce the Java form of the algebra expression.

Turning a query into an algebra expression

Getting the algebra expression for a Query is simply a matter of passing the parsed Query object to the transaction function in the `Algebra` class:

```
Query query = QueryFactory.create(.....) ;
Op op = Algebra.compile(query) ;
```

And back again.

```
Query query = OpAsQuery.asQuery(op) ;
System.out.println(query.serialize()) ;
```

This reverse translation can handle any algebra expression originally from a SPARQL Query, but not any algebra expression. It is possible to create programmatically useful algebra expressions that can not be truned into a query, especially if they involve algebra. Also, the query produced may not be exactly the same but will yield the same results (for example, filters may be moved because the SPARQL query algebra translation in the SPARQL specification moves filter expressions around).

Directly reading and writing algebra expression

The SSE class is a collection of functions to parse SSE expressions for the SPARQ algebra but also RDF terms, filter expressions and even dataset and graphs.

```
Op op = SSE.parseOp("(bgp (?s ?p ?o))") ;    // Read a string

Op op = SSE.readOp("filename.sse") ;     // Read a file
```

The SSE class simply calls the appropriate builder operation from the `com.hp.hpl.jena.sparql.sse.builder` package.

To go with this, there is a collection of writers for many of the Java structures in ARQ.

```
Op op = ... ;
SE.write(op) ;      // Write to stdout
```

Writers default to writing to `System.out` but support calls to any output stream (it manages the conversion to UTF-8) and ARQ own `IndentedWriter`s form for embedding in structured output. Again, SSE is simply passing the calls to the writer operation from the `com.hp.hpl.jena.sparql.sse.writer` package.

Creating an algebra expression programmatically

See the example in `src-examples/arq.examples.AlgebraExec`.

To produce the complete javadoc for ARQ, download an ARQ distribution and run the ant task 'javadoc-all'.

Evaluating a algebra expression

See the example in `src-examples/arq.examples.AlgebraExec`.

```
QueryIterator qIter = Algebra.exec(op,graph) ;

QueryIterator qIter = Algebra.exec(op,datasetGraph) ;
```

Evaluating an algebra expression produces a iterator of query solutions (called Bindings).

```
for ( ; qIter.hasNext() ; )
{
   Binding b = qIter.nextBinding() ;
   Node n = b.get(var_x) ;
   System.out.println(var_x+" = "+FmtUtils.stringForNode(n)) ;
}
qIter.close() ;
```

Operations of `CONSTRUCT`, `DESCRIBE` and `ASK` are done on on top of algebra evaluation. Applications can access this functionality by creating their own `QueryEngine` (see `arq.examples.engine.MyQueryEngine`) and it's factory. A query engine is a one-time use object for each query execution.

ARQ - Querying Remote SPARQL Services

SPARQL is a query language and a remote access protocol. The remote access protocol can be used with plain HTTP or over SOAP.

See Joseki for an implementation of an RDF publishing server, using the SPARQL protocol (HTTP and SOAP). Joseki uses ARQ to provide SPARQL query access to Jena models, including Jena persistent models.

ARQ includes a query engine capable of using the HTTP version. A version using SOAP is include in Joseki.

From your application

The `QueryExecutionFactory` has methods for creating a `QueryExecution` object for remote use. `QueryExecutionFactory.sparqlService`

These methods build a query execution object that uses the query engine in `com.hp.hpl.jena.sparql.engine.http`.

The remote request is made when the `execSelect`, `execConstruct`, `execDescribe` or `execAsk` method is called.

The results are held locally after remote execution and can be processed as usual.

From the command line

The `arq.sparql` command can issue remote query requests using the `--service` argument:

```
java -cp ... arq.query --service 'http://host/service' 'SELECT ?s WHERE {?s [] []}'
```

This takes a URL that is the service location.

The query given is parsed locally to check for syntax errors before sending.

Firewalls and Proxies

Don't forget to set the proxy for Java if you are accessing a public server from behind a blocking firewall. Most home firewalls do not block outgoing requests; many corporate firewalls do block outgoing requests.

If, to use your web browser, you need to set a proxy, you need to do so for a Java program.

Simple examples include:

```
-DsocksProxyHost=YourSocksServer

-DsocksProxyHost=YourSocksServer -DsocksProxyPort=port
```

```
-Dhttp.proxyHost=WebProxy -Dhttp.proxyPort=Port
```

This can be done in the application *if it is done before any network connection are made*:

```
System.setProperty("socksProxyHost", "socks.corp.com");
```

Consult the Java documentation for more details. Searching the web is also very helpful.

ARQ - Extending Query Execution

This page describes the mechanisms that can be used to extend and modify query execution within ARQ. Through these mechanisms, ARQ can be used to query different graph implementations and to provide different query evaluation and opti-mization strategies for particular circumstances. These mechanisms are used by TDB- and SDB.-

ARQ can be extended in various ways to incorporate custom code into a query. Custom filter functions and property functions p-rovi-de ways to add application specific code. The free text search capabilities, using Apache Lucene, are provided via a property function. Custom filter functions and property functions should be used where possible.

Jena itself can be extended by providing a new implementation of the `Graph` interface. This can be used to encapsulate specific specialised storage and also for wrapping non-RDF sources to look like RDF. There is a common implementation framewo-fk provided-fby `GraphBase` so only one operation, the `find` method, needs to be written for a read-only data source. Basic find works well is many cases, and the whole Jena API will be able to use the extension. For higher SPARQL performance, ARQ can be extended at the basic graph matching or algebra level.

Applications writers who extend ARQ at the query execution level should be prepared to work with the source code for ARQ for specific details and for finding code to reuse. Some example can be found in the `src-examples` directory in the ARQ download.

- Overview of ARQ Query processing-
- The Main Query Engine
- Graph matching and a custom StageGenerator
- OpExecutor
- Quads
- Mixed Graph Implementation Datasets
- Custom Query Engines
- Extend the algebra

Overview of ARQ Query Processing

The sequence of actions performed by ARQ to perform a query are parsing, algebra generation, execution building, high-level optimization, low-level optimization and finally evaluation. It is not usual to modify the parsing step nor the conversion from the parse tree to the algebra form, which is a fixed algorithm defined by the SPARQL standard. Extensions can modify the algebra form by transforming it from one algebra expression to another, including introducing new operators. See also the documentation on working with the SPARQL algebra in ARQ including building algebra expressions programmatically, rather than obtaining them from a query string.

Parsing

The parsing step turns a query string into a `Query` object. The class `Query` represents the abstract syntax tree (AST) for the query and provides methods to create the AST, primarily for use by the parser. The query object also provides methods to serialize the query to a string. Because this is the AST, the string produced is very close to the original query with the same syntactic elements, but without comments, and formatted with a whitespace for readability. It is not usually the best way to build a query programmatically and the AST is not normally an extension point.

The query object can be used many times. It is not modified once created, and in particular it is not modified by query execution.

Algebra generation

ARQ generates the SPARQL algebra expression for the query. After this a number of transformations can be applied (for example, identification of property functions) but the first step is the application of the algorithm in the SPARQL specification for translating a SPARQL query string, as held in a `Query` object into a SPARQL algebra expression. This includes the process of removing joins involving the identity pattern (the empty graph pattern).

For example, the query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?nick
WHERE  { ?x foaf:name  ?name ;
            foaf:mbox  ?mbox .
         OPTIONAL { ?x  foaf:nick  ?nick }
       }
```

becomes

```
(prefix ((foaf: <http://xmlns.com/foaf/0.1/>))
  (project (?name ?mbox ?nick)
    (leftjoin
      (bgp
        (triple ?x foaf:name ?name)
        (triple ?x foaf:mbox ?mbox)
      )
      (bgp (triple ?x foaf:nick ?nick)
      )
    )))
```

using the SSE syntax to write out the internal data-structure for the algebra.

The online SPARQL validator at sparql.org can be used to see the algebra expression for a SPARQL query. This validator is also include in Fuseki.

High-Level Optimization and Transformations

There is a collection of transformations that can be applied to the algebra, such as replacing equality filters with a more efficient graph pattern and an assignment. When extending ARQ, a query processor for a custom storage layout can choose which optimizations are appropriate and can also provide its own algebra transformations.

A transform is code that converts an algebra operation into other algebra operations. It is applied using

the `Transformer` class:

```
Op op = ... ;
Transform someTransform = ... ;
op = Transformer.transform(someTransform, op) ;
```

The `Transformer` class applies the transform to each operation in the algebra expression tree. `Transform` itself is an interface, with one method signature for each operation type, returning a replacement for the operator instance it is called on.

One such transformation is to turn a SPARQL algebra expression involving named graphs and triples into one using quads. This transformation is performed by a call to `Algebra.toQuadForm`.

Transformations proceed from the bottom of the expression tree to the top. Algebra expressions are best treated as immutable so a change made in one part of the tree should result in a copy of the tree above it. This is automated by the `TransformCopy` class which is the commonly used base class for writing transforms. The other helper base class is `TransformBase`, which provides the identify operation (returns the node supplied) for each transform operation.

Operations can be printed out in SSE syntax. The Java `toString` method is overridden to provide pretty printing and the static methods in `WriterOp` provide output to various output objects like `java.io.OutputStream`.

Low-Level Optimization and Evaluation

The step of evaluating a query is the process of executing the algebra expression, as modified by any transformations applied, to yield a stream of pattern solutions. Low-level optimizations include choosing the order in which to evaluate basic graph patterns. These are the responsibility of the custom storage layer. Low-level optimization can be carried out dynamically as part of evaluation.

Internally, ARQ uses iterators extensively. Where possible, evaluation of an operation is achieved by feeding the stream of results from the previous stage into the evaluation. A common pattern is to take each intermediate result one at a time (use `QueryIterRepeatApply` to be called for each binding) , substituting the variables of pattern with those in the incoming binding, and evaluating to a query iterator of all results for this incoming row. The result can be the empty iterator (one that always returns false for `hasNext`). It is also common to not have to touch the incoming stream at all but merely to pass it to sub-operations.

Query Engines and Query Engine Factories

The steps from algebra generation to query evaluation are carried out when a query is executed via the `QueryExecution.execSelect` or other `QueryExecution` exec operation. It is possible to carry out storage-specific operations when the query execution is created. A query engine works in conjunction with a `QueryExecution` created by the `QueryExecutionFactory` to provide the evaluation of a query pattern. `QueryExecutionBase` provides all the machinery for the different result types and does not need to be modified by extensions to query execution.

ARQ provides three query engine factories; the main query engine factory, one for a reference query engine and one to remotely execute a query. SDB and TDB provide their own query engine factories which they register during sub-system initialization. Both extend the main query engine described below.

The reference query engine is a direct top-down evaluation of the expression. It's purpose is to be simple so it can be easily verified and checked then its results used to check more complicated processing in the main engine and other implementations. All arguments to each operator are fully evaluated to produce intermediate in-memory tables then a simple implementation of the operator is called to calculate the results. It does not scale and does not perform any optimizations. It is intended to be clear and simple; it is not designed to be efficient.

Query engines are chosen by referring to the registry of query engine factories.

```
public interface QueryEngineFactory
{
    public boolean accept(Query query, DatasetGraph dataset, Context context) ;
    public Plan create(Query query, DatasetGraph dataset, Binding inputBinding,
Context context) ;

    public boolean accept(Op op, DatasetGraph dataset, Context context) ;
    public Plan create(Op op, DatasetGraph dataset, Binding inputBinding, Context
context) ;
}
```

When the query execution factory is given a dataset and query, the query execution factory tries each registered engine factory in turn calling the `accept` method (for query of algebra depending on how it was presented). The registry is kept in reverse registration order - the most recently registered query engine factory is tried first. The first query engine factor to return true is chosen and no further engine factories are checked.

When a query engine factory is chosen, the `create` method is called to return a `Plan` object for the execution. The main operation of the plan interface is to get the `QueryIterator` for the query.

See the example in `src-examples/arq.examples.engine.MyQueryEngine`.

The Main Query Engine

The main query engine can execute any query. It contains a number of basic graph pattern matching implementations including one that uses the `Graph.find` operation so it can work with any implementation of the Jena Graph SPI. The main query engine works with general purpose datasets but not quad stores directly; it evaluates patterns on each graph in turn. The main query engine includes optimizations for the standard Jena implementation of in-memory graphs.

High-level optimization is performed by a sequence of transformations. This set of optimizations is evolving. A custom implementation of a query engine can reuse some or all of these transformations (see `Algebra.optimize` which is the set of transforms used by the main query engine).

The main query engine is a streaming engine. It evaluates expressions as the client consumes each query solution. After preparing the execution by creating the initial conditions (a partial solution of one row and no bound variables or any initial bindings of variables), the main query engine calls `QC.execute` which is the algorithm to execute a query. Any extension that wished to reuse some of the main query engine by providing it's own `OpExecutor` must call this method to evaluate a sub-operation.

`QC.execute` finds the currently active `OpExecutor` factory, creates an `OpExecutor` object and invokes it to evaluate one algebra operation.

There are two points of extension for the main query engine:

- Stage generators, for evaluating basic graph patterns and reusing the rest of the engine.
- `OpExecutor` to execute any algebra operator specially.

The standard `OpExecutor` invokes the stage generator mechanism to match a basic graph pattern.

Graph matching and a custom StageGenerator

The correct point to hook into ARQ for just extending basic graph pattern matching (BGPs) is to provide a custom `StageGenerator`. (To hook into filtered basic graph patterns, the extension will need to provide it's own `OpExecutor` factory). The advantage of the `StageGenerator` mechanism, as compared to the more general `OpExecutor` described below, is that it more self-contained and requires less detail about the internal evaluation of the other SPARQL algebra operators. This extension point corresponds to section 12.6 "Extending SPARQL Basic Graph Matching".

Below is the default code to match a BGP from `OpExecutor.execute(OpBGP, QueryIterator)`. It merely calls fixed code in the `StageBuilder` class.The input is a stream of results from earlier stages. The execution must return a query iterator that is all the possible ways to match the basic graph pattern for each of the inputs in turn. Order of results does not matter.

```
protected QueryIterator execute(OpBGP opBGP, QueryIterator input)
{
    BasicPattern pattern = opBGP.getPattern() ;
    return StageBuilder.execute(pattern, input, execCxt) ;
}
```

The `StageBuilder` looks for the stage generator by accessing the context for the execution:

```
StageGenerator stageGenerator = (StageGenerator)context.get(ARQ.stageGenerator) ;
```

where the context is the global context and any query execution specific additions together with various execution control elements.

A `StageGenerator` is an implementation of:

```
    public interface StageGenerator
    {
        public QueryIterator execute(BasicPattern pattern,
                                     QueryIterator input,
                                     ExecutionContext execCxt) ;
    }
```

Setting the Stage Generator

An extension stage generator can be registered on a per-query execution basis or (more usually) in the global context.

```
    StageBuilder.setGenerator(Context, StageGenerator)
```

The global context can be obtained by a call to `ARQ.getContext()`

```
    StageBuilder.setGenerator(ARQ.getContext(), myStageGenerator) ;
```

In order to allow an extensions to still permit other graphs to be used, stage generators are usually chained, with each new custom one passing the execution request up the chain if the request is not

supported by this custom stage generator.

```
public class MyStageGenerator implements StageGenerator
{
    StageGenerator above = null ;

    public MyStageGenerator (StageGenerator original)
    { above = original ; }

    @Override
    public QueryIterator execute(BasicPattern pattern, QueryIterator input,
ExecutionContext execCxt)
    {
        Graph g = execCxt.getActiveGraph() ;
        // Test to see if this is a graph we support.
        if ( ! ( g instanceof MySpecialGraphClass ) )
            // Not us - bounce up the StageGenerator chain
            return above.execute(pattern, input, execCxt) ;
        MySpecialGraphClass graph = (MySpecialGraphClass )g ;
        // Create a QueryIterator for this request
     ...
```

This is registered by setting the global context (`StageBuilder` has a convenience operation to do this):

```
  // Get the standard one.
  StageGenerator orig = (StageGenerator)ARQ.getContext().get(ARQ.stageGenerator) ;
  // Create a new one
  StageGenerator myStageGenerator= new MyStageGenerator(orig) ;
  // Register it
  StageBuilder.setGenerator(ARQ.getContext(), myStageGenerator) ;
```

Example: `src-examples/arq.examples.bgpmatching`.

OpExecutor

A `StageGenerator` provides matching for a basic graph pattern. If an extension wishes to take responsibility for more of the evaluation then it needs to work with `OpExecutor`. This includes evaluation of filtered basic graph patterns.

An example query using a filter:

```
PREFIX  dc:   <http://purl.org/dc/elements/1.1/>
PREFIX  books: <http://example.org/book/>

SELECT  *
WHERE
  { ?book  dc:title  ?title .
    FILTER regex(?title, "Paddington")
  }
```

results in the algebra expression for the pattern:

```
    (filter (regex ?title "Paddington")
        (bgp (triple ?book dc:title ?title)
        ))
```

showing that the filter is being applied to the results of a basic graph pattern matching.

Note: this is not the way to provide custom filter operations. See the documentation for application-provided filter functions.

Each step of evaluation in the main query engine is performed by a `OpExecutor` and a new one is created from a factory at each step. The factory is registered in the execution context. The implementation of a specialized `OpExecutor` can inherit from the standard one and override only those algebra operators it wishes to deal with, including inspecting the execution and choosing to passing up to the super-class based on the details of the operation. From the query above, only regex filters might be specially handled.

Registering an `OpExecutorFactory`:

```
OpExecutorFactory customExecutorFactory = new MyOpExecutorFactory(...) ;
QC.setFactory(ARQ.getCOntext(), customExecutorFactory) ;
```

QC is a point of indirection that chooses the execution process at each stage in a query so if the custom execution wishes to evaluate an algebra operation within another operation, it shoudl call `QC.execute`. Be careful not to loop endlessly if the operation is itself handled by the custom evaluator. This can be done by swapping in a different `OpExecutorFactory`.

```
   // Execute an operation with a different OpExecution Factory

   // New context.
   ExecutionContext ec2 = new ExecutionContext(execCxt) ;
   ec2.setExecutor(plainFactory) ;

   QueryIterator qIter = QC.execute(op, input, ec2) ;

   private static OpExecutorFactory plainFactory =
      new OpExecutorFactory()
      {
         @Override
         public OpExecutor create(ExecutionContext execCxt)
         {
             // The default OpExecutor of ARQ.
             return new OpExecutor(execCxt) ;
         }
      } ;
```

Quads

If a custom extension provides named graphs, then it may be useful to execute the quad form of the query. This is done by writing a custom query engine and overriding `QueryEngineMain.modifyOp`:

```
  @Override
  protected Op modifyOp(Op op)
  {
     // Cope with initial bindings.
     op = Substitute.substitute(op, initialInput) ;
     // Use standard optimizations.
     op = super.modifyOp(op) ;
     // Turn into quad form.
     op = Algebra.toQuadForm(op) ;
```

```
    return op ;
}
```

The extension may need to provide its own dataset implementation so that it can detect when queries are directed to its named graph storage. TDB and SDB are examples of this.

Mixed Graph Implementation Datasets

The dataset implementation used in normal operation does not work on quads but instead can provide a dataset with a collection of graphs each from different implementation sub-systems. In-memory graphs can be mixed with database backed graphs as well as custom storage systems. Query execution proceeds per-graph so that an custom `OpExecutor` will need to test the graph to work with to make sure it is of the right class. The pattern in the `StageGenerator` extension point is an example of design pattern in that situation.

Custom Query Engines

A custom query engine enables an extension to choose which datasets it wishes to handle. It also allows the extension to intercept query execution during the setup of the execution so it can modify the algebra expression, introduce it's own algebra extensions, choose which high-level optimizations to apply and also transform to the expression into quad form. Execution can proceed with the normal algorithm or a custom `OpExecutor` or a custom Stage Generator or a combination of all three extension mechanism.

Only a small, skeleton custom query engine is needed to intercept the initial setup. See the example in `src-examples/arq.examples.engine.MyQueryEngine`.

While it is possible to replace the entire process of query evaluation, this is a substantial endeavour. `QueryExecutionBase` provides the machinery for result presentation (`SELECT`, `CONSTRUCT`, `DESCRIBE`, `ASK`), leaving the work of pattern evaluation to the custom query engine. `QueryExecutionFactory` assumes that `QueryExecutionBase` will be used.

Algebra Extensions

New operators can be added to the algebra using the `OpExt` class as the super-class of the new operator. They can be inserted into the expression to be evaluated using a custom query engine to intercept evaluation initialization. When evaluation of a query requires the evaluation of a sub-class of `OpExt`, the `eval` method is called. SDB uses this to introduce an operator that is implemented in SQL.

TDB

TDB is a component of Jena for RDF storage and query. It support the full range of Jena APIs. TDB can be used as a high performance RDF store on a single machine. This documentation describes the latest version, unless otherwise noted.

A TDB store can be accessed and managed with the provided command line scripts and via the Jena API.

See also Fuseki for a SPARQL server that uses TDB for persistent storage and provides the SPARQL

protocols for query, update and REST update over HTTP.

Documentation

- TDB Download and Installation
- TDB Requirements
- Command line utilities
- Using TDB from Java through the API
- Transactions
- Assemblers for Graphs and Datasets
- Datasets and Named Graphs
- Dynamic Datasets: Query a subset of the named graphs
- Quad filtering: Hide information in the dataset
- The TDB Optimizer
- TDB Configuration
- Value Canonicalization
- TDB Design
- Use on 64 bit or 32 bit Java systems

```
Hi Sarven,
I think I have identified the problem.

With Lucene we can have only one IndexWriter at the time.
When we run larq.larqbuilder and we specify --desc=tdb.ttl
we need to make sure tdb.ttl does not have a ja:textIndex
property in it.

This is because larqbuilder creates one Lucene IndexWriter
and then it calls the DataSourceAssembler which is trying
to create another Lucene IndexWriter if ja:textIndex is
there.

Also, from the fact that you still have "null" in your
error message... I am not 100% sure you are using the
latest ARQ SNAPSHOT. To be absolutely sure, could you
try deleting it from your .m2 Maven repository.

However, I am experiencing another problem with Fuseki
which I do not understand:

cd /tmp
svn co http://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/trunk/ fuseki
cd /tmp/fuseki
mvn test

-------------------------------------------------------
  T E S T S
-------------------------------------------------------
Running org.openjena.fuseki.TS_Fuseki
  INFO [qtp1548452562-20] (SPARQL_ServletBase.java:118) - [1] POST
http://localhost:3535/dataset/update
  INFO [qtp1548452562-20] (SPARQL_ServletBase.java:153) - [1] 204 No Content
  INFO [qtp1548452562-21] (SPARQL_ServletBase.java:118) - [2] GET
http://localhost:3535/dataset/data?default=
```

```
  INFO [qtp1548452562-21] (SPARQL_ServletBase.java:153) - [2] 200 OK
  INFO [qtp1548452562-22] (SPARQL_ServletBase.java:118) - [3] GET
http://localhost:3535/dataset/data?graph=http://graph/1
  INFO [qtp1548452562-22] (SPARQL_ServletBase.java:155) - [3] 404 No such graph:
<http://graph/1>
  INFO [qtp1548452562-23] (SPARQL_ServletBase.java:118) - [4] POST
http://localhost:3535/dataset/update
  INFO [qtp1548452562-23] (SPARQL_ServletBase.java:153) - [4] 204 No Content
  INFO [qtp1548452562-24] (SPARQL_ServletBase.java:118) - [5] GET
http://localhost:3535/dataset/data?graph=http://graph/1
  INFO [qtp1548452562-24] (SPARQL_ServletBase.java:155) - [5] 404 No such graph:
<http://graph/1>
  INFO [qtp1548452562-19] (SPARQL_ServletBase.java:118) - [6] POST
http://localhost:3535/dataset/update
  INFO [qtp1548452562-19] (SPARQL_ServletBase.java:153) - [6] 204 No Content


It stops here, forever.

To retry the patch for LARQ in Fuseki, do:

cd /tmp
svn co http://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/trunk/ fuseki
cd /tmp/fuseki
wget
https://issues.apache.org/jira/secure/attachment/12482758/JENA-
63_Fuseki_r1136050.patch
patch -p0 < JENA-63_Fuseki_r1136050.patch
mvn -DskipTests=true package

Then you should be able to index a dataset using:

java -cp target/fuseki-0.2.1-SNAPSHOT-sys.jar larq.larqbuilder
--allow-duplicates --larq=/tmp/lucene --desc=/path/to/your/assembler.ttl

But, please, make sure you do not have any ja:textIndex in your assembler.ttl
when you do the initial bulk indexing.

You should put the ja:textIndex back in for normal operations.

It's not ideal, but at least now I understand the cause of the problem
and there is a workaround.

Let me know how it goes.

Paolo

Sarven Capadisli wrote:
> Thanks a lot Paulo. Just some feedback on these steps:
>
> My pom.xml contains http://pastebin.com/5HAJBL95
>
> $ java -cp target/fuseki-0.2.1-SNAPSHOT-sys.jar larq.larqbuilder
> --allow-duplicates --larq=/usr/lib/fuseki/lucene-index/
> --desc=/usr/lib/fuseki/tdb2.ttl
>
> 11:56:34 WARN  DataSourceAssembler  :: Unable to initialize LARQ using
> org.apache.jena.larq.assembler.AssemblerLARQ: null
> 11:56:34 WARN  DataSourceAssembler  :: Unable to initialize LARQ using
```

> com.hp.hpl.jena.query.larq.AssemblerLARQ: null
>
> -Sarven
>
> On Thu, 2011-06-16 at 10:45 +0100, Paolo Castagna wrote:
>> Hi Sarven,
>> first of all, thanks for your email.
>>
>> This is about an open issue (an improvement) which aim is to add the new LARQ
>> (i.e. the one as separate module) to Fuseki and make it as easy as possible for
>> people to use.
>>
>> See: https://issues.apache.org/jira/browse/JENA-63
>> The issue is still open and there are problems.
>> I cleaned up the attachments on JENA-63 and uploaded a new patch for Fuseki.
>>
>> This is how you can apply the patch to Fuseki:
>>
>> cd /tmp
>> svn co http://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/trunk/ fuseki
>> cd /tmp/fuseki
>> wget
>> https://issues.apache.org/jira/secure/attachment/12482758/JENA-
63_Fuseki_r1136050.patch
>> patch -p0 < JENA-63_Fuseki_r1136050.patch
>> mvn package
>>
>> Then you should be able to index a dataset using:
>>
>> java -cp target/fuseki-0.2.1-SNAPSHOT-sys.jar larq.larqbuilder
>> --allow-duplicates --larq=/tmp/lucene --desc=/path/to/your/assembler.ttl
>>
>> However, there is a problem (I improved the error message):
>>
>> 10:32:35 WARN DataSourceAssembler :: Unable to initialize LARQ using
>> org.apache.jena.larq.assembler.AssemblerLARQ: Lock obtain timed out:
>> NativeFSLock@/tmp/lucene/write.lock
>>
>> This is the stack trace:
>>
>> java.lang.reflect.InvocationTargetException
>>   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
>>   at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
>>   at
>>
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:2
5)
>>   at java.lang.reflect.Method.invoke(Method.java:597)
>>   at
>>
com.hp.hpl.jena.sparql.core.assembler.DataSourceAssembler.createTextIndex(DataSourc
eAssembler.java:115)
>>   at
>>
com.hp.hpl.jena.sparql.core.assembler.DataSourceAssembler.createTextIndex(DataSourc
eAssembler.java:97)
>>   at
>>

```
com.hp.hpl.jena.sparql.core.assembler.DatasetAssembler.open(DatasetAssembler.java:2
2)
>>      at
>>
com.hp.hpl.jena.assembler.assemblers.AssemblerGroup$PlainAssemblerGroup.openBySpeci
ficType(AssemblerGroup.java:118)
>>      at
>>
com.hp.hpl.jena.assembler.assemblers.AssemblerGroup$PlainAssemblerGroup.open(Assemb
lerGroup.java:105)
>>      at
>>
com.hp.hpl.jena.assembler.assemblers.AssemblerGroup$ExpandingAssemblerGroup.open(As
semblerGroup.java:69)
>>      at
com.hp.hpl.jena.assembler.assemblers.AssemblerBase.open(AssemblerBase.java:37)
>>      at
com.hp.hpl.jena.assembler.assemblers.AssemblerBase.open(AssemblerBase.java:34)
>>      at
>>
com.hp.hpl.jena.sparql.core.assembler.AssemblerUtils.build(AssemblerUtils.java:88)
>>      at arq.cmdline.ModAssembler.create(ModAssembler.java:55)
>>      at
arq.cmdline.ModDatasetAssembler.createDataset(ModDatasetAssembler.java:31)
>>      at arq.cmdline.ModDataset.getDataset(ModDataset.java:22)
>>      at larq.larqbuilder.exec(larqbuilder.java:84)
>>      at arq.cmdline.CmdMain.mainMethod(CmdMain.java:85)
>>      at arq.cmdline.CmdMain.mainRun(CmdMain.java:47)
>>      at arq.cmdline.CmdMain.mainRun(CmdMain.java:34)
>>      at larq.larqbuilder.main(larqbuilder.java:50)
>> Caused by: org.apache.lucene.store.LockObtainFailedException: Lock obtain timed
>> out: NativeFSLock@/tmp/lucene/write.lock
>>      at org.apache.lucene.store.Lock.obtain(Lock.java:84)
>>      at org.apache.lucene.index.IndexWriter.<init>(IndexWriter.java:1097)
>>      at
org.apache.jena.larq.IndexWriterFactory.create(IndexWriterFactory.java:36)
>>      at org.apache.jena.larq.assembler.AssemblerLARQ.make(AssemblerLARQ.java:85)
>>      ... 21 more
>>
>>
>> It seems to me that Lucene is failing to acquire the write.lock,
>> as if initialization code were called twice.
>>
>> I have not yet identified the cause of this and I am investigating.
>>
>> Apologies and be patience (until we make progress and we close JENA-63).
>>
>> Paolo
>>
>> Sarven Capadisli wrote:
>>> Hi,
>>>
>>> I'd like to get Fuseki and LARQ running. Below is where I'm at. Any help
>>> would be great:
>>>
>>> I use https://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/trunk
>>> and it sits at /usr/lib/fuseki
>>>
>>> I have
```

```
>>> http://ftp.heanet.ie/mirrors/www.apache.org/dist//lucene/java/3.2.0/lucene-
3.2.0.tgz
at /usr/lib/lucene/
>>>
>>> I've applied
>>> https://issues.apache.org/jira/secure/attachment/12478735/JENA-
63_Fuseki_r8810.patch
>>>
>>> My /usr/lib/fuseki/pom.xml is http://pastebin.com/Cpaz75ai
>>>
>>> My /usr/lib/fuseki/tdb2.ttl is http://pastebin.com/SXv5LWEn
>>>
>>> When I run
>>> $java -cp target/fuseki-0.2.1-SNAPSHOT-sys.jar larq.larqbuilder
>>> --allow-duplicates --larq=/usr/lib/lucene/index/
>>> --desc=/usr/lib/fuseki/tdb2.ttl
>>>
>>> I get http://pastebin.com/JQPqsPtH
>>>
>>> -Sarven
>>>
>
```

Hi Tao,
could you try this for me:

svn co http://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/trunk/
fuseki cd fuseki

edit pom.xml adding LARQ dependency:

```
     <dependency>
       <groupId>org.apache.jena</groupId>
       <artifactId>jena-larq</artifactId>
       <version>1.0.0-incubating</version>
     </dependency>
```

```
mvn clean package
java -jar target/jena-fuseki-0.2.2-incubating-SNAPSHOT-server.jar
--config=config.ttl
```

```
---- config.ttl ----
@prefix :        <#> .
@prefix fuseki:  <http://jena.apache.org/fuseki#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix tdb:     <http://jena.hpl.hp.com/2008/tdb#> .
@prefix ja:      <http://jena.hpl.hp.com/2005/11/Assembler#> .

[] rdf:type fuseki:Server ;
   fuseki:services (
     <#service1>
   ) .

<#service1> rdf:type fuseki:Service ;
    fuseki:name                     "ds" ;
    fuseki:serviceQuery             "sparql" ;
    fuseki:serviceQuery             "query" ;
    fuseki:serviceUpdate            "update" ;
```

```
    fuseki:serviceUpload               "upload" ;
    fuseki:serviceReadWriteGraphStore  "data" ;
    fuseki:serviceReadGraphStore       "get" ;
    fuseki:serviceReadGraphStore       "" ;
    fuseki:dataset                     <#dataset1> ;
    .

<#dataset1> rdf:type      tdb:DatasetTDB ;
    tdb:location "/tmp/tdb" ;
    ja:textIndex "/tmp/lucene"
    .
----
```

I get results when I query with:
s-query --service=http://127.0.0.1:3030/ds/sparql "PREFIX pf:
<http://jena.hpl.hp.com/ARQ/property#> SELECT * { ?s pf:textMatch 'word'}
LIMIT 10"

Now, I have done the exactly the same with:
http://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/tags/jena-fuseki
-0.2.1-incubating-RC-1/
and
http://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/tags/jena-fuseki
-0.2.1-incubating/
and I experience your problem. (I checked both since I was not 100% which
tag corresponds to the actual release, it should be
jena-fuseki-0.2.1-incubating,
right?)

The Lucene index is created correctly and it's there, I can query in using
the larq.larq command line:
java -cp target/jena-fuseki-0.2.1-incubating-server.jar larq.larq
--larq=/tmp/lucene "word"

The only hypothesis I have is that somehow something goes wrong with the
assembler which initialize LARQ. I am sorry not being able to be more
helpful at the moment.

The only suggestion I have for you is to try using Fuseki from trunk:
http://svn.apache.org/repos/asf/incubator/jena/Jena2/Fuseki/trunk/
and let me know if you experience problems with that. That works for me.

In the meantime, I'll see if I can find what's the cause of this problem.

Paolo

Tao (陶信东) wrote:
> Hi Paolo,
> The index files seem ok (see below). It was obviously created by
> Fuseki (It didn't exist before I create it). It seems that LARQ is not
> called when performing the pf:textMatch search.
>
> -rw-r--r-- 1   17459784 Apr 27 14:32 _2.fdt
> -rw-r--r-- 1   2325668 Apr 27 14:32 _2.fdx
> -rw-r--r-- 1   40 Apr 27 14:32 _2.fnm
> -rw-r--r-- 1   2269211 Apr 27 14:32 _2.frq
> -rw-r--r-- 1   581420 Apr 27 14:32 _2.nrm
> -rw-r--r-- 1   1414550 Apr 27 14:32 _2.prx
> -rw-r--r-- 1   40894 Apr 27 14:32 _2.tii

> -rw-r--r-- 1   4475539 Apr 27 14:32 _2.tis
> -rw-r--r-- 1   253 Apr 27 14:32 segments_1
> -rw-r--r-- 1   20 Apr 27 14:32 segments.gen
> -rw-r--r-- 1   0 Apr 27 14:32 write.lock
>
> -----Original Message-----
> From: Paolo Castagna [mailto:castagna.li...@googlemail.com]
> Sent: Friday, April 27, 2012 3:26 PM
> To: jena-users@incubator.apache.org
> Subject: Re: [ANN] Release of Apache Jena LARQ 1.0.0-incubating
>
> Hi Tao,
> can you share the bit of your Fuseki configuration where you point at
> the path for your Lucene indexes?
>
> #dataset> rdf:type tdb:DatasetTDB ;
>   ...
>   ja:textIndex "/path/to/lucene/index/" ;
>   .
>
> Can you do an ls -la of that directory to see the file sizes?
>
> Can you try to point to a non existing directory and restart Fuseki?
>
> If the directory exists an empty index will be created (this is a side
> effect of opening a Lucene Directory when your Lucene index does not
exist).
> If the directory does not exist your dataset will be indexed.
>
> Let's see if you can sort this out and let me know if you have ideas
> on how to improve this.
>
> Paolo
>

> Tao (陶信东) wrote:
>> Thanks Paolo. Now I can find the lucene files. But pf:textMatch still
>> doesn't seem to work. See details below.
>>
>> select * where {?s rdfs:label ?o} limit 10
>> --
>> -
>> ------
>> -
>> | s | o
>> |
>> ==
>> =
>> ======
>> =
>> | <http://xmlns.com/foaf/0.1/LabelProperty> | "Label
> Property"
>> |
>> | <http://xmlns.com/foaf/0.1/Person> | "Person"
>> |
>> | <http://www.w3.org/2000/10/swap/pim/contact#Person> | "Person"
>> |
>> | <http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing> | "Spatial
Thing"
>> |

```
>> | <http://xmlns.com/foaf/0.1/Document>                    | "Document"
>> |
>> | <http://xmlns.com/foaf/0.1/Organization>                | "Organization"
>> |
>> | <http://xmlns.com/foaf/0.1/Group>                       | "Group"
>> |
>> | <http://xmlns.com/foaf/0.1/Agent>                       | "Agent"
>> |
>> | <http://xmlns.com/foaf/0.1/Project>                     | "Project"
>> |
>> | <http://xmlns.com/foaf/0.1/Image>                       | "Image"
>> |
>> ------------------------------------------------------------------------
>> -
>> ------
>> -
>>
>> prefix pf: <http://jena.hpl.hp.com/ARQ/property#>
>> select * where {?s pf:textMatch "+Label"} limit 10
>> -----
>> | s |
>> =====
>> -----
>>
>>
>> -----Original Message-----
>> From: Paolo Castagna [mailto:castagna.li...@googlemail.com]
>> Sent: Thursday, April 26, 2012 10:26 PM
>> To: jena-users@incubator.apache.org
>> Subject: Re: [ANN] Release of Apache Jena LARQ 1.0.0-incubating
>>

>> Tao (陶信东) wrote:
>>> Thanks Paolo, I'll try that.
>>>
>>> Meanwhile, would you please clarify whether Fuseki will build LARQ
>>> index automatically, after the patch?
>> Yes, if the directory does not exist LARQ will create it and build
>> the Lucene index automatically, if a directory exists a Lucene index
>> will be opened pointing to that directory (however, if the directory
>> exists and it's empty, you'll have an empty Lucene index as result):
>>
>>         if ( indexPath != null )
>>         {
>>             File path = new File(indexPath) ;
>>             if ( !path.exists() )
>>             {
>>                 log.debug("Directory {} does not exist, building
>> Lucene
>> index...") ;
>>                 path.mkdirs() ;
>>                 build ( dataset, path ) ;
>>             }
>>             directory = FSDirectory.open(path);
>>         } else {
>>             directory = new RAMDirectory();
>>         }
>>
>> http://svn.apache.org/repos/asf/incubator/jena/Jena2/LARQ/tags/jena-l
>> a
```

>> rq-1.0
>> .0-incubating/src/main/java/org/apache/jena/larq/assembler/AssemblerL
>> A
>> RQ.jav
>> a
>>
>>> Your reply and Venkat's seemed different regarding this.
>> This is because things have been improved since the latest release.
>>
>> Paolo
>>
>>> Thanks
>>> Tao
>>>
>>> -----Original Message-----
>>> From: Paolo Castagna [mailto:castagna.li...@googlemail.com]
>>> Sent: Wednesday, April 25, 2012 6:51 PM
>>> To: jena-users@incubator.apache.org
>>> Subject: Re: [ANN] Release of Apache Jena LARQ 1.0.0-incubating
>>>
>>> Hi Tao
>>>

>>> Tao(陶信东) wrote:
>>>> Hi Paolo,
>>>>
>>>> This is great! I've been waiting for it since several weeks ago.
>>> Good to know.
>>>
>>>> However, after I configured Fuseki for it by adding LARQ to my
>>>> class path and the following lines to the assembly file,
>>>> pf:textMatch doesn't work (the usual sparql query works).
>>>>
>>>> <#dataset> rdf:type tdb:DatasetTDB ;
>>>>   tdb:location "/path/to/my/tdb/indexes/" ;
>>>>   ja:textIndex "/path/to/lucene/index/" ;
>>>>   .
>>>>
>>>> Is Fuseki supported by this LARQ release?
>>> Fuseki does not include LARQ.
>>>
>>> For more details, please, see also:
>>> https://issues.apache.org/jira/browse/JENA-63
>>> https://issues.apache.org/jira/browse/JENA-164
>>>
>>> The good news is that it is extremely easy to check out Fuseki
>>> sources, patch the pom.xml file and repackage it (i.e. mvn package).
See:
>>> https://issues.apache.org/jira/secure/attachment/12504042/JENA-63_Fu
>>> s
>>> e
>>> ki_r12
>>> 03107.patch
>>> (be warned, that patch might not be up-to-date, but you see it's
>>> just a new dependency on LARQ).
>>>
>>>> Or should I build the lucene index
>>>> by myself when Fuseki started?
>>> No, you do not need that.
>>>

>>> 1. Checkout Fuseki source code.
>>> 2. Add LARQ dependency to the Fuseki pom.xml file 3. mvn package
>>>
>>> Let me know how it goes and if you have problems.
>>>
>>> Cheers,
>>> Paolo
>>>
>>>> Thanks
>>>> Tao
>>>>
>>>> -----Original Message-----
>>>> From: Paolo Castagna [mailto:castagna.li...@googlemail.com]
>>>> Sent: Tuesday, April 24, 2012 5:43 AM
>>>> To: jena-users@incubator.apache.org
>>>> Subject: [ANN] Release of Apache Jena LARQ 1.0.0-incubating
>>>>
>>>> LARQ 1.0.0-incubating has been released, this is the first release
>>>> of LARQ as separate module and under the Apache License.
>>>>
>>>> LARQ is a combination of ARQ and Lucene aimed at providing
>>>> developers with the ability to perform free text searches within
>>>> their SPARQL
>>> queries.
>>>> Documentation for LARQ is available here:
>>>>
>>>> - http://incubator.apache.org/jena/documentation/larq/
>>>> - http://incubator.apache.org/jena/documentation/javadoc/larq/
>>>>
>>>>
>>>> == Mailing lists
>>>>
>>>> The user mailing list for Jena is jena-users@incubator.apache.org
>>>> Send email to jena-users-subscr...@incubator.apache.org to subscribe.
>>>>
>>>> See also:
>>>> http://incubator.apache.org/jena/help_and_support/index.html
>>>>
>>>>
>>>> == About This Release
>>>>
>>>> The main new feature in this release is support for updating the
>>>> Lucene index as RDF statements are added/removed to the Jena Model
>>>> via the Jena APIs.
>>>> Moreover, LARQ now depends on Apache Lucene 3.5.0.
>>>>
>>>>
>>>> == Download
>>>>
>>>> Maven artifacts are here:
>>>> http://repo1.maven.org/maven2/org/apache/jena/jena-larq/1.0.0-incub
>>>> a
>>>> t
>>>> i
>>>> ng/
>>>>
>>>> Source release is here:
>>>> http://www.apache.org/dyn/closer.cgi/incubator/jena/jena-larq-1.0.0
>>>> -

```
>>>> i
>>>> n
>>>> cubati
>>>> ng
>>>>
>>>>
>>>> == Status
>>>>
>>>> Apache Jena is an effort undergoing incubation at the Apache
>>>> Software Foundation (ASF), sponsored by the Apache Incubator PMC.
>>>>
>>>> Incubation is required of all newly accepted projects until a
>>>> further review indicates that the infrastructure, communications,
>>>> and decision making process have stabilized in a manner consistent
>>>> with other successful ASF projects.
>>>>
>>>> While incubation status is not necessarily a reflection of the
>>>> completeness or stability of the code, it does indicate that the
>>>> project has yet to be fully endorsed by the ASF.
>>>>
>>>> For more information about the incubation status of the Jena
>>>> project you can go to the following page:
>>>> http://incubator.apache.org/projects/jena.html
>>>>
>
```

#summary Install Fuseki triple store with LARQ text index

= Introduction =

Jena Fuseki is a SPARQL server and triple store which can be used as a backend for ONKI Light. Enabling the LARQ index speeds up text search queries. This currently requires performing a custom build of Fuseki, because LARQ is not included in the standard Fuseki distribution.

= Preparation =

Building Fuseki requires a Java SDK and Maven. Checking out the latest source requires Subversion. You can install these on Ubuntu like this: `sudo apt-get install openjdk-6-jdk maven subversion`

If you have previously used Maven, you may want to clear your Maven repository before building Fuseki to get rid of any old package versions: `rm -rf ~/.m2`

= Checking out the source =

Check out the Jena source: `svn co https://svn.apache.org/repos/asf/jena/trunk/ Jena`

= Adding LARQ dependency =

You will need to add a LARQ dependency to the Jena/jena-fuseki/pom.xml file. Edit the file and add this block after the other dependencies:

{{{
  <dependency>
    <groupId>org.apache.jena</groupId>
    <artifactId>jena-larq</artifactId>
    <version>1.0.1-SNAPSHOT</version>
  </dependency>
}}}

Newer versions of LARQ may be available by the time you read this. Check out [https://repository.apache.org/content/groups/snapshots/org/apache/jena/jena-larq/ this directory listing].

= Building =

Build the package using Maven like this:
{{{
cd Jena/jena-fuseki
mvn clean package
}}}

= Installing =

After a succesful build, the `target/` subdirectory should contain a file jena-fuseki-X.X.X-SNAPSHOT-distribution.tar.gz (among others). This tarball can be installed just like the standard Fuseki distribution, see the [http://jena.apache.org/documentation/serving_data/index.html Fuseki installation instructions]. In short, you just unpack the tarball into a directory and run the `fuseki-server` command to start the server.

= Configuring LARQ indexes =

To actually use LARQ indexes, you will need to specify a filesystem path for them in a Fuseki configuration file. Here is an example configuration file which uses a TDB store at `/tmp/tdb` and a LARQ Lucene index at `/tmp/lucene`:

{{{
@prefix :   <#> .
@prefix fuseki: <http://jena.apache.org/fuseki#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

```
@prefix tdb:     <http://jena.hpl.hp.com/2008/tdb#> .
@prefix ja:      <http://jena.hpl.hp.com/2005/11/Assembler#> .

[] rdf:type fuseki:Server ;
   fuseki:services (
     <#service1>
   ) .

<#service1> rdf:type fuseki:Service ;
    fuseki:name                     "ds" ;
    fuseki:serviceQuery             "sparql" ;
    fuseki:serviceQuery             "query" ;
    fuseki:serviceUpdate            "update" ;
    fuseki:serviceUpload            "upload" ;
    fuseki:serviceReadWriteGraphStore  "data" ;
    fuseki:serviceReadGraphStore       "get" ;
    fuseki:serviceReadGraphStore       "" ;
    fuseki:dataset                  <#dataset1> ;
    .

[] ja:loadClass "com.hp.hpl.jena.tdb.TDB" .
tdb:DatasetTDB  rdfs:subClassOf  ja:RDFDataset .
tdb:GraphTDB    rdfs:subClassOf  ja:Model .

<#dataset1> rdf:type      tdb:DatasetTDB ;
    tdb:location "/tmp/tdb" ;
    ja:textIndex "/tmp/lucene" .

}}}
```

Save this as `larq-config.ttl` and then you can run Fuseki with `./fuseki-server --config=larq-config.ttl`

= Updating LARQ indexes =

Currently, LARQ indexes are not kept up to date when the data in the triple store changes (see the [https://issues.apache.org/jira/browse/JENA-164 JENA-164 issue] for current status). To update the indexes, you can do either of these:

 # Shut down Fuseki, remove the Lucene index directory, and start Fuseki again. The index will be rebuilt when the server starts.
 # Shut down Fuseki and run the larqbuilder script that comes with Fuseki, e.g. like this:

{{{
java -cp fuseki-server.jar larq.larqbuilder --larq=/tmp/lucene --desc=fuseki-config.ttl
}}}
Then start Fuseki again.

Introduction

Jena Fuseki is a SPARQL server and triple store which can be used as a backend for ONKI Light. Enabling the LARQ index speeds up text search queries. This currently requires performing a custom build of Fuseki, because LARQ is not included in the standard Fuseki distribution.

Preparation

Building Fuseki requires a Java SDK and Maven. Checking out the latest source requires Subversion. You can install these on Ubuntu like this: `sudo apt-get install openjdk-6-jdk maven subversion`

If you have previously used Maven, you may want to clear your Maven repository before building Fuseki to get rid of any old package versions: `rm -rf ~/.m2`

Checking out the source

Check out the Jena source: `svn co https://svn.apache.org/repos/asf/jena/trunk/ Jena`

Adding LARQ dependency

You will need to add a LARQ dependency to the Jena/jena-fuseki/pom.xml file. Edit the file and add this block after the other dependencies:

```
<dependency>
  <groupId>org.apache.jena</groupId>
  <artifactId>jena-larq</artifactId>
  <version>1.0.1-SNAPSHOT</version>
</dependency>
```

Newer versions of LARQ may be available by the time you read this. Check out this directory listing.

Building

Build the package using Maven like this:

```
cd Jena/jena-fuseki
mvn clean package
```

Installing

After a succesful build, the `target/` subdirectory should contain a file jena-fuseki-X.X.X-SNAPSHOT-distribution.tar.gz (among others). This tarball can be installed just like the standard Fuseki distribution, see the Fuseki installation instructions. In short, you just unpack the tarball into a directory and run the `fuseki-server` command to start the server.

Configuring LARQ indexes

To actually use LARQ indexes, you will need to specify a filesystem path for them in a Fuseki configuration file. Here is an example configuration file which uses a TDB store at `/tmp/tdb` and a LARQ Lucene index at `/tmp/lucene`:

```
@prefix :        <#> .
@prefix fuseki:  <http://jena.apache.org/fuseki#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix tdb:     <http://jena.hpl.hp.com/2008/tdb#> .
@prefix ja:      <http://jena.hpl.hp.com/2005/11/Assembler#> .
[] rdf:type fuseki:Server ;
   fuseki:services (
     <#service1>
   ) .
<#service1> rdf:type fuseki:Service ;
    fuseki:name                     "ds" ;
    fuseki:serviceQuery             "sparql" ;
    fuseki:serviceQuery             "query" ;
    fuseki:serviceUpdate            "update" ;
    fuseki:serviceUpload            "upload" ;
    fuseki:serviceReadWriteGraphStore  "data" ;
    fuseki:serviceReadGraphStore       "get" ;
    fuseki:serviceReadGraphStore       "" ;
    fuseki:dataset                  <#dataset1> ;
    .
[] ja:loadClass "com.hp.hpl.jena.tdb.TDB" .
tdb:DatasetTDB  rdfs:subClassOf  ja:RDFDataset .
tdb:GraphTDB    rdfs:subClassOf  ja:Model .
<#dataset1> rdf:type      tdb:DatasetTDB ;
    tdb:location "/tmp/tdb" ;
    ja:textIndex "/tmp/lucene" .
```

Save this as `larq-config.ttl` and then you can run Fuseki with `./fuseki-server --config=larq-config.ttl`

Updating LARQ indexes

Currently, LARQ indexes are not kept up to date when the data in the triple store changes (see the JENA-164 issue for current status). To update the indexes, you can do either of these:

1. Shut down Fuseki, remove the Lucene index directory, and start Fuseki again. The index will be rebuilt when the server starts.
2. Shut down Fuseki and run the larqbuilder script that comes with Fuseki, e.g. like this:

```
java -cp fuseki-server.jar larq.larqbuilder  --larq=/tmp/lucene --desc=fuseki-
config.ttl
```

Then start Fuseki again.