sa.engine Java Interfaces

Stream Analyze Sweden AB
Sweden

Version 2.2
2020-09-08
`sa_JavaAPI_2.2.pdf`

This document describes the external interfaces between sa.engine and the programming language Java. There are mainly two ways to interface sa.engine with Java programs: In the *client* interface Java programs call sa.engine, while in the *plugin* interface foreign OSQL functions are implemented as Java methods. The combination is also possible where foreign functions in Java call sa.engine back through the client interface.

Table of contents

1. Introduction

There are two main kinds of external interfaces, the *client* and the *plugin* interfaces:

- With the *client interface* a program in Java calls sa.engine. The client interface allows OSQL queries and function calls to be shipped from application programs to *remote* sa.engine servers or to an *embedded* sa.engine system running in the same process as the application.

- With the *plugin interface* OSQL functions are implemented as public methods in Java. The foreign Java functions are executed in the same process as sa.engine. The client interface can be used also in foreign Java function implementations.

The result of an OSQL query or function call is an *object stream*, which is a possibly infinite stream of objects. The client interface provides primitives to map over the elements in such object streams. Analogously the plugin interface provides API primitives to produce object streams.

This documentation describes the sa.engine API for Java 8 where the new "lambda-functions" enable a very elegant object stream interface where lambda functions in Java are used as callback functions. It is possible to use sa.engine also for older Java versions by using lower level interface primitives not described here.

This documentation introduces the Java API of sa.engine through a number of example programs whose source codes are in the folders `sa.engine/demo/*/Java` of an installed sa.engine system. In that folder you will find a number of examples for how to use the Java API. You are assumed to be familiar with OSQL.

To compile an example Java program using the Java API of sa.engine you must make sure that your CLASSPATH includes the file `sa.engine/bin/sa_Java.jar` that implements the API. If sa.engine is installed and CLASSPATH set correctly you can compile the Java program `Hello.java` in folder `sa.engine/demo/Hello/Java` on your PC with:

```
javac Hello.java
```

2. The client interface

With the *client interface* there are two ways to call sa.engine from Java:

- In the *query interface* strings containing OSQL statements are sent to sa.engine for dynamic evaluation. The result from a query is an object stream. The `forEach` interface

in Java 8 provides a powerful mechanism where the system iterates over object streams and applies user lambda-functions (callback methods) for each element in the received object stream. The embedded query interface is relatively slow since the OSQL statements must be parsed and compiled at run time.

- In the *function interface* sa.engine functions are directly called from Java, without the overhead of dynamically parsing and executing OSQL statements. The result of a function call is an object stream. The function interface is significantly faster than the query interface. It is therefore recommended to always define sa.engine functions stored in the local database for the various sa.engine operations performed by the application and then use the function interface to invoke them directly.

When calling sa.engine from application programs, the application usually often runs as a client to an sa.engine server (SAS) running on some other computer. This is called the *remote connection*. With the remote connection several applications running in different locations can remotely access the same SAS concurrently. The Java applications and the SAS run as separate programs so that the server will survive client crashes and vice versa.

It is also possible to run the system as an *embedded* sa.engine system in the same process and memory address space as the client application. Several client threads can thereby concurrently access the embedded sa.engine thread. This is called the *local connection*.

2.1. Connections

A Java object of class `Connection` represents connections to an sa.engine server. The connection is established when the connection object is created using the constructors:

```
new Connection(String p)
new Connection()
```

The constructor `Connection(p)` will establish a *remote connection* to an sa.engine *peer* named `p` that can be i) an sa.engine client running on an edge device, ii) an sa.engine *stream server* (SAS) coordinating communication with edge devices, or iii) a *nameserver*, which is a SAS that keeps track of all other peers in a federation of sa.engine peers. The format of the string `p` is one of:

```
peer
peer@host
peer@host:portno
```

If just a `peer` name is specified the peer must be a local peer known by the nameserver running on the same computer as the client. The local nameserver can beo reached using the peer name `nameserver`. If `peer@host` is specified a connection is established to a peer running on the name server of the specified `host`. Specifying `peer@localhost` is equivalent to just `peer`. The nameserver by default listens on port 35021; the format `peer@host:portno` is used when the nameserver on that host uses some other port.

When the connection constructor has no argument, a *local connection* is established to an embedded sa.engine system running in the same main memory process as the caller.

2.2. The query interface

In the *query interface*, strings being OSQL statements are sent to sa.engine stream servers or edges for execution. The following is a Java application that prints the five first natural number in file `sa.engine/demo/clent/Java/QueryRange.java`:

```
import com.sa.callin.*;
public class QueryRange {
    public static void main(String argv[]) {
        Connection c = new Connection("p");
        ObjectStream s = c.query("range(1,5)");
        s.mapAll(e -> System.out.println(e.getIntElem(0)));
    }
}
```

Example 1. Mapping over the result from a query

The constructor `Connection("p")` opens a connection `c` to peer named **`"p"`** managed by the local nameserver running on the same host as the application. The peer **`"p"`** can also be an *edge device* managed by the nameserver or a some SAS coordinated by the nameserver that manages the edge device.

The method call `c.query(q)` sends a query string `q` to be executed by the peer to which `c` connects. The query will thereby be compiled and optimized by sa.engine and then an *object stream* `s` of class `ObjectStream` is constructed by sa.engine to represent the result stream of the query. The application can utilize the *lambda experssions* of Java 8 (https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) to map over object streams using the `mapAll` method. In the example a lambda expression is appplied on all elements `e` in the result object stream from the OSQL query **`range(1,5)`**.

An object stream element `e` is a tuple of one or several objects. In the example each `e` is a tuple containing a single integer. To access the object in position `pos` of a tuple `e` is the method `e.GetIntElem(pos)` is called. The tuple positions are enumerated from 0 and up.

The example shows how execute a single OSQL query where the system will block the current thread to wait for tuples to arrive on the object stream. Several OSQL queries can be executed in parallel in separate Java threads.

A common case is that one or several OSQL statements are sent to the server for synchronous evaluation and then immediately waiting for them to finish. This is made by applying the method

`run()` on the result stream from method `query`. For example:

```
Tuple res = c.query("create function add1(Number x) -> Number as x + 1").run();
```

Here the method `run()` iterates over the entire result stream to execute the `create` statement. It returns the *last* element of the object stream when the query is completed. In the example the object stream contains only the object representing the function `add1`, so it will be returned from `run()`. The method `run()` cannot be used for queries returning infinite streams (Sec. 2.4), as that would cause an infinite loop.

2.3. The function interface

The time to dynamically compile and optimize a query by sa.engine can be rather long, so a better way is to directly call OSQL functions through the *function interface*. The following Java code in file `sa.engine/demo/client/Java/CallRange.java` calls the OSQL function `range(1,5)`:

```
import com.sa.callin.*;
public class CallRange {
    public static void main(String argv[]) {
        Connection c = new Connection("p");
        Tuple argl = Tuple.make(1, 5);
        ObjectStream s = c.call("range", argl);
        s.mapAll (e -> System.out.println(e.getIntElem(0)));
    }
}
```

Example 2. Calling a function and mapping over the result

Most of the code is the same as in `QueryRange.java` in Sec. 2.2. The only difference is that an *argument tuple* (1,5) bound to `argl` is first constructed by calling the variadic Java method `Tuple.make(1,5)` (Sec 4.1.1) and that the function named **`"range"`** with arguments in tuple `argl` is called by method `c.call("range", argl)`. The argument tuple `argl` contains arguments of OSQL function calls from Java. An object stream is returned by method call `c.call()` on which a lambda expression to print all result elements `e` is applied.

2.4. Mapping over infinite streams

The results of queries and function calls returning streams are also represented as object streams. There is no upper limit on how many elements can be retrieved from such an object stream. The system is able to handle object streams `s` containing indefinite numbers of elements, in which case the mapping will never terminate. In such cases the mapping will have to be done with the method `map` where the lambda expression returns a Boolean *continuation flag* that terminate the mapping when `false`. Alternatively, the mapping can be terminated by throwing an exception.

For example, the following query in `sa.engine/demo/client/Java/CallHeartbeat.java` prints the numbers 0.0 and 0.1 by iterating over the two first elements of the indefinite stream returned by `heartbeat(0.1)`:

```
import com.sa.callin.*;
public class CallHeartbeat {
    public static void main(String argv[]) {
        Connection c = new Connection("p");
        Tuple argl = Tuple.make(0.01);
        ObjectStream s = c.call("heartbeat", argl);
        int[] cnt = {1};
        s.map(e -> {
            System.out.println(e.getDoubleElem(0));
            return cnt[0]++ < 2;
        });
    }
}
```

Example 3. Example on how to stop an object stream

When the lambda expression returns the continuation flag `true` while the mapping continues, and `false` when it is terminated. In the example it is terminated after printing the two first elements `e` of the object stream. Notice that lambda expressions in Java 8 require the counter `cnt` to be called by reference by defining it as an array.

3. Implementing foreign Java functions

Foreign functions in Java are defined as methods in public classes in a Java class files that are dynamically loaded into sa.engine. This chapter describes through examples the different kinds of foreign functions definable in Java.

3.1. A Hello World foreign function

As a first very simple example, we implement a foreign Java function *hello() -> Charstring* that returns the string "Hello World". The file `sa.engine/demo/Hello/Java/Hello.java` has the following contents:

```
import com.sa.callin.*;
import com.sa.callout.*;
public class Hello {
    public void helloF(CallContext cxt, Tuple tpl) {
        tpl.setElem(0,"Hello World");
        cxt.emit(tpl);
    }
}
```

Example 4. Simplest foreign function in Java returning the string "Hello World"

The parameter `cxt` is a *context* object of class `CallContext` for managing the call by sa.engine, and `tpl` is a *parameter tuple* of class `Tuple` representing both the arguments (inputs) and results (outputs) of the foreign function call.

In the example the function binds the unbound *parameter position* 0 of the parameter tuple `tpl` to the desired result string **`"Hello World"`** by calling the method `tpl.setElem(0, "Hello World")`. Positions of tuples are enumerated 0 and up; in this case there is no argument so the result is in position 0.

When the result has been filled in using `tpl.setElem()` the entire tuple `tpl` is sent to sa.engine using the method `cxt.emit(tpl)`.

When `Hello.java` is compiled with the command

```
make compile
```

you can start the console REPL with the shell command `sa.engine` and there register the external Java function implementation by executing the OSQL statement:

```
create function hello() -> Charstring
  as foreign 'JAVA:Hello/helloF';
```

The function can then be called immediately:

```
hello();
```

You will notice that, if needed, sa.engine first dynamically loads a Java Virtual Machine (JVM) to execute the call. The call will fail if no JVM is found. If it fails to load the JVM, make sure that a suitable JVM can be reached from the folder where the console REPL was started. If the JVM cannot be found, set the variable JAVA_HOME properly.

The OSQL script in `sa.engine/demo/Hello/validate.osql` validates that Hello World works correctly. It is recommended that you make such test scripts for all new foreign functions. You can run the test script with

```
make test
```

The foreign Java function definitions are permanently saved in the database image when saved on disk so that they will be reloaded when sa.engine is started again with the saved image.

Once Hello World works you know that you have set up the environment for compiling and plugging in Java code to sa.engine correctly. You are ready to define any foreign Java function.

3.2. Foreign function with arguments and result

The arguments of a foreign OSQL function in Java are stored in the parameter tuple as well as the results. For example, the function *myconcat(Charstring x, Charstring y) -> Charstring* concatenates strings *x* and *y*. It has the following implementation as method `myconcatBBF` in `sa.engine/demo/Basic/Java/Simple.c`:

```
public void myconcatBBF (CallContext cxt, Tuple tpl) {
   String x = tpl.getStringElem(0); // Pick up 1st argument x
   String y = tpl.getStringElem(1); // Pick up 2nd argument x
   tpl.setElem(2, x+y);
   cxt.emit(tpl);
 }
```

Example 5. Foreign Java function implementation taking two parameters

In this case the arguments `x` and y of `myconcat(x,y)` are in positions zero and one of the parameter tuple `tpl` and the computed result is bound in position two.

Compile the Java code in folder `sa.engine/demo/Basic/Java/` with

```
make compile
```

The OSQL code of *myconcat()* in file `sa.engine/demo/Basic/Java/definitions.osql` is:

```
create function myconcat(Charstring x, Charstring y) -> Charstring
  as foreign 'JAVA:Simple/myconcatBBF';
```

The OSQL script `sa.engine/demo/Basic/validate.osql` includes validation of *myconcat()*. The code in the Java folder can be tested with:

```
make test
```

3.3. Foreign function with several results

Foreign functions can also return more than a single value as a tuple. For example, the foreign function *sqrt2(Number x)->(Number pos,Number neg)* returns both the positive and negative square roots of number *x*. It is implemented as method `sqrt2BFF` in file `sa.engine/demo/Basic/Java/Simple.java`:

```
public void sqrt2BFF(CallContext cxt, Tuple tpl) {
   double root = Math.sqrt(tpl.getDoubleElem(0));
   tpl.setElem(1, root);
   tpl.setElem(2, -root);
   cxt.emit(tpl);
 }
```

Example 6. Foreign Java function implementation returning tuple of several values

In this case the foreign function has a single input argument in position 0 of tuple `tpl`. The method call `tpl.getDoubleElem(0)` converts the input argument into a floating point number. An exception is raised if the argument is not a number. The method `tpl.setElem` is called twice to set the two result elements in the parameter tuple.

The OSQL code of *sqrt2()* is:

```
create function sqrt2(Number x) -> (Number neg, Number pos)
  as foreign 'JAVA:Simple/sqrt2BFF';
```

3.4. Foreign function returning a vector

The OSQL type `Vector` is represented in Java as arrays. The function *vsqrt2(Number x) -> Vector of Number* returns the negative and positive square root of number *x*. It is implemented as method `vsqrt2BF` in file `sa.engine/demo/Basic/Java/Simple.java`:

```
public void vsqrt2BF (CallContext cxt, Tuple tpl) {
  double x = tpl.getDoubleElem(0); // Pick up argument x
  if(x >= 0) {
      double root = Math.sqrt(x);
      double[] roots = {-root, root};
      tpl.setElem(1, roots);
      cxt.emit(tpl);
}
```

Example 7. Foreign function method returning a vector

In this case the foreign function has a single input argument in position 0 of the parameter tuple `tpl`. The method call `tpl.getDoubleElem(0)` converts the input argument into a floating point number. An exception is raised if the argument is not a number. The method `tpl.setElem` will convert the Java array root to an OSQL object of type `Vector`.

The OSQL definition of *vsqrt2()* is:

```
create function vsqrt2(Number x) -> Vector of Number
  as foreign 'JAVA:Simple/vsqrt2BF';
```

3.5. Foreign function taking vectors as arguments

As an example of a function taking vectors as arguments, the function *dotprod(Vector v, Vector w) ->Number* returns the Cartesian product of vectors *v* and *w*. It is implemented as method `dotprodBBF` in file `sa.engine/demo/Basic/Java/Collections.java` as:

```
public void dotprodBBF(CallContext cxt, Tuple tpl)
  {
      double[] v = tpl.getDoubleArrayElem(0); // First argument
      double[] w = tpl.getDoubleArrayElem(1); // Second argument
      double prod = 0;
      if (v.length! = w.length)
          throw new sa_Exception("Array index out of bounds");
      for(int i=0; i < v.length; i++) {
          prod = prod + v[i]*w[i];
      }
      tpl.setElem(2,prod);
      cxt.emit(tpl);
  }
}
```

Example 8. Foreign Java function implementation taking vectors as arguments

The method call `tpl.getDoubleArrayElem(pos)` returns the element at position `pos` of tuple `tpl` as a Java array.

The expression `throw new sa_Exception(msg)` raises an sa.engine exception.

The OSQL definition of *dotprod()* is:

```
create function dotprod(Vector v, Vector w) -> Number
  as foreign 'JAVA:Collections/dotprodBBF';
```

3.6. Foreign function generating a bag

Foreign functions can return bags of values. For example, the foreign function *natural(Number n)->Bag of Number* returns a bag of the integers from *m* to *n*. It is implemented as method `naturalBBF` in file `sa.engine/demo/Basic/Java/Collections.java`:

```
    public void naturalBBF(CallContext cxt, Tuple tpl) {
        int n = tpl.getIntElem(0);
        for(int i=1; i<=n; i++) {
            tpl.setElem(1, i);
            cxt.emit(tpl);
        }
    }
```

Example 9. Foreign Java function implementation returning a bag of numbers from m to n

The method call `tpl.getIntElem(0)` converts the input argument in position 0 of `tpl` to an integer. Notice that `cxt.emit(tpl)` is called several times to iteratively emit each element of the result bag of numbers.

The OSQL definition of *natural()* is:

```
create function natural(Number m, Number n) -> Bag of Number
  as foreign 'JAVA:Collections/naturalBBF';
```

3.7. Foreign aggregate function

To implement foreign aggregate functions in Java that iterate over collections (bags, streams or vectors) the method `CallContext.map` is used. For example, the function *sqsum(Bag of Number b)->Number* will sum up the square all numbers in bag *b*. It is implemented as method `sqsumBF` in file `sa.engine/demo/Aggregation/Java/Aggregation.java`:

```
   public void sqsumBF(CallContext cxt, Tuple tpl) {
      double[] sum = {0};
      cxt.mapAll(tpl.getOidElem(0), e -> {
            double x = e.getDoubleElem(0);
            sum[0] += x*x;
      });
      tpl.setElem(1, sum[0]);
      cxt.emit(tpl);
   }
```

Example 10. Implementation of an aggregate function jsum over a bag of numbers

If you call `sqsum(range(1,10))` from the console REPL you will get the result 55.

The method `CallContext.mapAll(coll, l)` takes as arguments a collection `coll` to map over and a lambda expression `l`. The expression `l` is applied on each element `e` in the result object stream.

The OSQL definition of *sqsum()* is:

```
create function sqsum(Bag b) -> Number
  as foreign 'JAVA:Aggregation/sqsumBF';
```

3.8. Aggregation over vectors

A foreign aggregate function implementation over bags can also be used for aggregating over vectors. For example, the method `sqsumBF` above can also be used for computing the sum of the square of numbers in vector by defining the function:

```
create function sqsum(Vector v) -> Number
  as foreign 'JAVA:Aggregation/sqsumBF';
```

3.9. Aggregation over finite streams

A foreign aggregate function implementation over bags can also be used for aggregating over finite streams. For example, the method `sqsumBF` above can also be used for computing the sum of the square of numbers in vector by defining the function:

```
create function sqsum(Stream v) -> Number
  as foreign 'JAVA:Aggregation/sqsumBF';
```

3.10. Foreign function returning a record

Records in sa.engine are represented as class `Record` in Java (Sec. 4.3). For example, the function *rsqrt2(Number x)->Record* returns the square root of *x* as a record `{"neg": -r "pos": r}`. It is implemented as method `rsqrt2BF` in file `sa.engine/demo/Basic/Java/Collections.java`:

```
    public void rsqrt2BF(CallContext cxt, Tuple tpl) {
        double x = tpl.getDoubleElem(0);
        if(x >= 0) {
            double root = Math.sqrt(x);
            // Construct record {"neg":-root,"pos":root}:
            Record r = new Record();
            r.put("neg", -root);
            r.put("pos", root);
            tpl.setElem(1,r);
            cxt.emit(tpl);           // Emit record r
        }
```

Example 11. Foreign function returning a record.

The method call `r.put(a,v)` sets the value for attribute `a` in in record `r` to `v`.

The OSQL definition of *rsqrt2()* is:

```
create function rsqrt2(Number x) -> Record
  as foreign 'JAVA:Collections/rsqrt2BF';
```

3.11. Foreign function accessing a record

The function *getnum(Record r,Charstring a)->Number* returns attribute a in record r as a number. It is implemented as method `getnumBBF` in file `sa.engine/demo/Basic/Java/Collections.java`:

```
    public void getnumBBF(CallContext cxt, Tuple tpl) {
            Record r = tpl.getRecordElem(0);
            String field = tpl.getStringElem(1);
            Tuple val = r.get(field);
            if(val == null) return; // Returns nil
            double x = val.getDoubleElem(0);
            tpl.setElem(2, x);
            cxt.emit(tpl);
        }
```

Example 12. Foreign function accessing a record.

The OSQL definition of *getnum()* is:

```
create function getnum(Record r, Charstring field) -> Number
  as foreign 'JAVA:Collections/getnumBBF';
```

3.12. Foreign function returning an infinite stream

The same mechanism as for bags is used for returning (possibly infinite) streams of values. For example, the function *natural_numbers()->Stream of Number* returns an infinite stream of the natural numbers (integers from one to infinity). It is implemented as method `natural_numbersF` in file `sa.engine/demo/Streams/Java/Streams.java`:

```
public void natural_numbersF(CallContext cxt, Tuple tpl) {
    int i=0;
    while(true) {
        tpl.setElem(0, i++);
        cxt.emit(tpl);
    }
}
```

Example 13. Foreign Java function implementation returning an infinite stream of all natural numbers

If you call `natural_numbers()` from the console REPL an infinite stream of number is returned and the system will print natural numbers until you interrupt it with CTRL-C. The call `section(natural_numbers(), 10, 20)` will return a finite stream.

3.13. Foreign stream transformation function

A *stream transformation function* takes a stream as argument and produces a new transformed stream as result. For example, the function *power_stream(Stream s,Number n) -> Stream of Number* generates a stream of x^n of the numbers *x* in stream *s*. It has the following implementation in `sa.engine/demo/Streams/Java/Streams.java`:

```
public void power_streamBBF(CallContext cxt, Tuple tpl) {
    double exp = tpl.getDoubleElem(1);
    cxt.mapAll(tpl.getOidElem(0), e -> {
        double x = e.getDoubleElem(0)
        tpl.setElem(2,Math.pow(x, exp));
        cxt.emit(tpl);
    });
}
```

Example 14. Implementation of a stream transformation function

The important thing here is that `emit()` is called for each stream element in the lambda expression.

The OSQL definition of *power_stream()* is:

```
create function power_stream(Stream of Number s, Number n) -> Stream of Number
  as foreign 'JAVA:Streams/power_streamBBF';
```

3.14. Exception handling

sa.engine can raise two kinds of runtime Java exceptions: `NoMoreData` and `sa_Exception`. The user need not declare or catch these runtime exceptions, but sometimes a `try ... finally ...` construct is needed in order to guarantee that resources allocated by the foreign function are always freed. The error message for an sa.engine exception can be obtained by calling the Java system exception method `getMessage()`.

The exception `NoMoreData` is raised when an application indicates that no more data is needed.

The exception `sa_Exception` is raised by the system when some error is detected. To raise a new exception with a new error message, `sa_Exception` has a constructor that takes an error string as parameter.

The cause of an `sa_Exception` can be investigated through the following `sa_Exception` attributes:

`errno`: Attribute holding the sa.engine error number. Only some of the system messages have error numbers. If `errno==-1` it indicates that the exception did not have a specific error number.

`errstr`: Attribute holding the error message string.

`errform`: Attribute holding the object causing the error.

3.15. Multi-directional foreign functions

Foreign OSQL functions can be made invertible. For example, assume a foreign square root function:

```
create function sqroot(Number x) -> Bag of (Number r)
  as foreign 'JAVA:Sqrt/sqrootBF';
```

Its definition if file `Sqrt.java` is:

```
import com.sa.callin.*;
import com.sa.callout.*;
public class Sqrt {
    public void sqrootBF(CallContext cxt, Tuple tpl) {
        double x = tpl.getDoubleElem(0);
        if(x==0) {
            tpl.setElem(1, 0);
            cxt.emit(tpl);
        }
        else if(x > 0) {
            double root = Math.sqrt(x);
            tpl.setElem(1, root);
            cxt.emit(tpl);
            tpl.setElem(1, -root);
            cxt.emit(tpl);
        }
    }
}
```

Example 15. Implementation of the foreign function sqroot(number x) -> bag of (Number r)

The bag of numbers 2.0 and -2.0 will be returned by the query `sqroot(4)`.

If `sqroot` were invertible we could also make a query calling its inverse:

```
select x from Number x where 2 in sqroot(x); /* Result is 4.0 */
```

Since the definition above is *not* multidirectional the system will raise an error that the query is not executable because variable `x` is not bound.

We now define `sqroot` as an invertible foreign Java function by redefining it as:

```
create function sqroot(Number x) -> Bag of (Number r)
  as multidirectional ('bf' foreign 'JAVA:Sqrt/sqrootBF')
                      ('fb' foreign 'JAVA:Sqrt/sqrootFB');
```

after first having adding the method `sqrootFB` to `Sqrt.java`:

```
  public void sqrootFB(CallContext cxt, Tuple tpl) {
      double x = tpl.getDoubleElem(1);
      tpl.setElem(0, x*x);
      cxt.emit(tpl);
  }
```

Example 16. The Java implementation of the inverse to the sqroot foreign function

The method `sqrootFB` implements the inverse of `sqroot`. In this case position 1 in `tpl` is bound and position 0 is filled in.

As in the example, multi-directional foreign functions are functions that can be executed also when arguments are unknown. The benefit of multi-directional foreign functions is that a larger class of queries calling the function is executable (safe), and that the system can make better query optimization.

A multi-directional foreign function has several implementations depending on the *binding pattern* of its arguments and results. The binding pattern is a string of 'b':s and 'f':s, indicating which arguments or results in a given implementation are known or unknown, respectively.
A simple foreign OSQL function is a special case where all arguments are known and all results are unknown.

To implement a multi-directional function you first need to think of for which binding patterns implementations are needed. In the `sqroot` case one implementation handles the two square roots and the other one handles the inverse square.

4. Data objects

The Java API uses a number of Java classes and methods documented in this section.

4.1. Tuples

The class `Tuple` is a commonly used class in the sa.engine Java interface. A tuple represents an ordered finite sequence of sa.engine objects.

Tuples are used for many purposes in the interfaces:

- Parameter tuples are used in the plugin interface for representing argument and results of methods implementing foreign functions in Java (Sec. 3).
- Tuples represent retrieved data elements when iterating through streams (Sec. 2).
- Tuples are used for representing argument lists in sa.engine function calls from applications (Sec. 2.3).

4.1.1. Tuple creation

A new tuple with size `s` is created with the constructor `Tuple(s)`.

If all elements of the new tuple are known in advance, you can also construct it using the variadic constructor `Tuple.make(…)`, for example `Tuple.make(1,2,3)`.

The method `t.getArity()` returns the number of elements in tuple `t`.

The elements of a tuple are enumerated starting at 0 and can be accessed through a number of *tuple access functions* specific for each element class, as described next.

4.1.2. Integer elements

To access an integer stored in position `pos` of tuple `t`, call the method `t.getIntElem(pos)`. Floating point numbers are rounded to the closest integer. An error is generated if there is no number in the specified position of the tuple.

To store integer `i` in element `pos` of tuple `t`, call the method `t.setElem(pos, i)`.

4.1.3. Floating point elements

To get a double precision floating point number stored in position `pos` of a tuple `t`, call the method `t.getDoubleElem(pos)`. Integers are converted to floating point numbers. An error is generated if there is no number in the specified position of the tuple.

To store floating point number `x` in element `pos` of tuple `t`, call the method `t.setElem(pos,x)`.

4.1.4. String elements

To get a string stored in position `pos` of tuple `t`, call the method `t.getStringElemt(pos)`. An error is generated if the element is not a string.

To store a string `str` in element `pos` of tuple `t`, call the method `t.setElem(pos,str)`.

4.1.5. Data type of element

The data type of element pos of tuple t can be tested with the following Boolean methods:

```
t.isDouble(pos)
t.isInteger(pos)
t.isObject(pos)
t.isString(pos)
t.isVector(pos)
t.isRecord(pos)
t.isTuple(pos)
```

4.2. Vectors

OSQL vectors (type `Vector`) are represented in Java arrays. To get an array stored in position `pos` of tuple `t`, call the method `t.getArrayElem(pos)`. An error is generated if there is no vector in the specified position of the tuple.

To store a copy of array `a` as an OSQL vector in element `pos` of tuple `t`, call the method `t.setElem(pos,a)`.

4.3. Records

OSQL records (type `Record`) are represented in Java as objects of class `Record`. To access a record stored in position `pos` of tuple `t`, call the method `t.getRecordElem(pos)`. An error is generated if there is no record in position `pos` of tuple `t`.

As for tuples there are a number of methods for accessing the element of a record depending on their type: `getBool, getInt, getLong, getDouble, getString, getSeq, getArray, getIntArray, getDoubleArray, getStringrray, getRecordArray, getOid, getBinary`, or `getRecord`.

Values are internally stored as single element tuples in a record. To retrieve the tuple `t` holding the value of key `k` in record `r` call `Tuple t = r.get(k)`. The call `r.getInt("k")` is thus equivalent to `r.get("k").getIntElem(0)`.

To store a copy of record `r` as an OSQL record in element `pos` of tuple `t`, call the method `t.setElem(pos,r)`.

To set value `v` to key `k` for record `r` call the method `r.put(k,v)`.

4.4. Object proxies

An *object proxy* is a Java object representing a corresponding referenced sa.engine object accessed through a connection. Object proxies can reference any kind of data stored in sa.engine, including numbers, strings, surrogate objects, arrays, and internal sa.engine data structures. Object proxies are represented using the Java class `Oid`.

To get an object proxy representing the OSQL object stored in position `pos` of tuple `t`, call the method `t.getElem(pos)`. If there is no element in position `pos`, Java `null` is returned.

To store an OSQL object proxy `o` in element `pos` of tuple `t`, call the method `t.setElem(pos,t)`.

The name of the sa.engine datatype of an object proxy `o` can be retrieved as a string by calling the method `o.getTypename()`.

The proxy object representing the datatype of proxy object `o` can be retrieved by calling the method `o.getType()`.

The connection to the sa.engine system owning an object proxy `o` can be retrieved by calling the method `o.getConnection()`.