



You are here: [Home](#) | [Enterprise Java](#) | Introduction to PostgreSQL PL/java



About Bear Giles



Introduction to PostgreSQL PL/java

by Bear Giles on October 21st, 2012 | Filed in: Enterprise Java Tags: Database, PostgreSQL

Modern databases allow stored procedures to be written in a variety of languages. One commonly implemented language is java.N.B., this article discusses the PostgreSQL-specific java implementation. The details will vary with other databases but the concepts will be the same.

Installation of PL/Java

Installation of PL/Java on an Ubuntu system is straightforward. I will first create a new template, *template_java*, so I can still create databases without the pl/java extensions.

At the command line, assuming you are a database superuser, enter

```
1 # apt-get install postgresql-9.1
2 # apt-get install postgresql-9.1-pljava-gcj
3
4 $ createdb template_java
5 $ psql -d template_java -c 'update db_database set datistemplate='t' where datnam='template_java''
6 $ psql -d template_java -f /usr/share/postgresql-9.1-pljava/install.sql
```

Limitations

The prepackaged Ubuntu package uses the Gnu GCJ java implementation, not a standard OpenJDK or Sun implementation. GCJ compiles java source files to native object code instead of byte code. The most recent versions of PL/Java are “trusted” – they can be relied upon to stay within their sandbox. Among other things this means that you can’t access the filesystem on the server.

If you must break the trust there is a second language, ‘javaU’, that can be used. Untrusted functions can only be created a the database superuser.

More importantly this implementation is single-threaded. This is critical to keep in mind if you need to communicate to other servers.

Something to consider is whether you want to compile your own commonly used libraries with GCJ and load them into the PostgreSQL server as shared libraries. Shared libraries go in */usr/lib/postgresql/9.1/lib* and I may have more to say about this later.

Quick verification

We can easily check our installation by writing a quick test function. Create a scratch database using *template_java* and enter the following SQL:

```
1 CREATE FUNCTION getsysprop(VARCHAR) RETURNS VARCHAR
2 AS 'java.lang.System.getProperty'
3 LANGUAGE java;
4
5 SELECT getsysprop('user.home');
```

You should get “/var/lib/postgresql” as a result.

Installing Our Own Methods

This is a nice start but we don’t really gain much if we can’t call our own methods. Fortunately it isn’t hard to add our own.

A simple PL/Java procedure is

```
01 package sandbox;
02
```

Newsletter



12,4
enjoy
comp
Join
exclu
news
well
Andr

other related technologies.
As an **extra bonus**, by join
brand new e-books, publ
Geeks and their JCG partn
pleasure!

Join Us



With
visitc
auth
the t
arou
the k
enco

```

03 public class PLJava {
04     public static String hello(String name) {
05         if (name == null) {
06             return null;
07         }
08         return 'Hello, ' + name + '!';
09     }
10 }
11 }

```

There are two simple rules for methods implementing PL/Java procedures:

- they must be public static
- they must return *null* if any parameter is *null*

That's it.

Importing the java class into PostgreSQL server is simple. Let's assume that the package classes are in */tmp/sandbox.jar* and our java-enabled database is *mydb*. Our commands are then

```

01 --
02 -- load java library
03 --
04 -- parameters:
05 -- url_path - where the library is located
06 -- url_name - how the library is referred to later
07 -- deploy - should the deployment descriptor be used?
08 --
09 select sqlj.install_jar('file:///tmp/sandbox.jar', 'sandbox', true);
10
11 --
12 -- set classpath to include new library.
13 --
14 -- parameters
15 -- schema - schema (or database) name
16 -- classpath - colon-separated list of url_names.
17 --
18 select sqlj.set_classpath('mydb', 'sandbox');
19
20 -- -----
21 -- other procedures --
22 -- -----
23
24 --
25 -- reload java library
26 --
27 select sqlj.replace_jar('file:///tmp/sandbox.jar', 'sandbox', true);
28
29 --
30 -- remove java library
31 --
32 -- parameters:
33 -- url_name - how the library is referred to later
34 -- undeploy - should the deployment descriptor be used?
35 --
36 select sqlj.remove_jar('sandbox', true);
37
38 --
39 -- list classpath
40 --
41 select sqlj.get_classpath('mydb');
42
43 --

```

It is important to remember to set the classpath. Libraries are automatically removed from the classpath when they're unloaded but they are NOT automatically added to the classpath when they're installed.

We aren't quite finished – we still need to tell the system about our new function.

```

01 --
02 -- create function
03 --
04 CREATE FUNCTION mydb.hello(varchar) RETURNS varchar
05 AS 'sandbox.PLJava.hello'
06 LANGUAGE java;
07
08 --
09 -- drop this function
10 --
11 DROP FUNCTION mydb.hello(varchar);
12
13 --

```

We can now call our java method in the same manner as any other stored procedures.

Deployment Descriptor

There's a headache here – it's necessary to explicitly create the functions when installing a library and dropping them when removing a library. This is time-consuming and error-prone in all but the simplest cases.

Fortunately there's a solution to this problem – deployment descriptors. The precise format is defined by ISO/IEC 9075-13:2003 but a simple example should suffice.

If you have a blog with unique content then you should check out our partners program. You can earn money for Java Code Geeks and help us in addition to utilizing our **review** service to **monetize** your technical website.

Carrer Opportunities

Software Engineer, Backend
FULL-TIME) March 22nd, 2014

JAVA Programmer/Developer
March 21st, 2014

Lead Java Backend Engineer
March 21st, 2014

Engineer Software 3 (Full-time)
2014

Java Applications Developer
– Tampa, FL (FULL-TIME)

Tags

Akka Android Tutorial Apache
Apache Maven Apache
Cloud Concurrency
Design Patterns Eclipse
Grails IDE Interview Java
JavaFX JAXB JBoss JUnit
JPA JSF JSON JSP
Logging MongoDB
Oracle GlassFish Performance
Project Management REST
Security Spring
Spring MVC

```

01 SQLActions[] = {
02     'BEGIN INSTALL
03         CREATE FUNCTION javatest.hello(varchar)
04         RETURNS varchar
05         AS 'sandbox.PLJava.hello'
06         LANGUAGE java;
07     'END INSTALL',
08     'BEGIN REMOVE
09         DROP FUNCTION javatest.hello(varchar);
10     'END REMOVE'
11 }

```

You must tell the deployer about the deployment descriptor in the jar's MANIFEST.MF file. A sample maven plugin is

```

01 <plugin>
02   <groupId>org.apache.maven.plugins</groupId>
03   <artifactId>maven-jar-plugin</artifactId>
04   <version>2.3.1</version>
05   <configuration>
06     <archive>
07       <manifestSections>
08         <manifestSection>
09           <name>postgresql.ddr</name> <!-- filename -->
10           <manifestEntries>
11             <SQLJDeploymentDescriptor>TRUE</SQLJDeploymentDescriptor>
12           </manifestEntries>
13         </manifestSection>
14       </manifestSections>
15     </archive>
16   </configuration>
17 </plugin>

```

The database will now know about our methods as they are installed and removed.

Internal Queries

One of the 'big wins' with stored procedures is that queries are executed on the server itself and are MUCH faster than running them through the programmatic interface. I've seen a process that required over 30 minutes via Java knocked down to a fraction of a second by simply moving the queried loop from the client to the server.

The JDBC URL for the internal connection is "jdbc:default:connection". You cannot use transactions (since you're within the caller's transaction) but you can use savepoints as long as you stay within a single call. I don't know if you can use CallableStatements (other stored procedures yet) – you couldn't in version 1.2 but the Ubuntu 11.10 package uses version 1.4.2.

Lists of scalar values are returned as *Iterators* in the java world and *SETOF* in the SQL world.

```

1 public static Iterator<String> colors() {
2     List<String> colors = Arrays.asList('red', 'green', 'blue');
3     return colors.iterator();
4 }

```

and

```

1 CREATE FUNCTION javatest.colors()
2 RETURNS SETOF varchar
3 AS 'sandbox.PLJava.colors'
4 IMMUTABLE LANGUAGE java;

```

I've added the *IMMUTABLE* keyword since this function will always return the same values. This allows the database to perform caching and query optimization.

You don't need to know the results, or even the size of the results, before you start. Following is a sequence that's believed to always terminate but this hasn't been proven. (Unfortunately I've forgotten the name of the sequence.) As a sidenote this isn't a complete solution since it doesn't check for overflows – a correct implementation should either check this or use *BigInteger*.

```

01 public static Iterator seq(int start) {
02     Iterator iter = null;
03     try {
04         iter = new SeqIterator(start);
05     } catch (IllegalArgumentException e) {
06         // should log error...
07     }
08     return iter;
09 }
10
11 public static class SeqIterator implements Iterator {
12     private int next;
13     private boolean done = false;
14
15     public SeqIterator(int start) {
16         if (start <= 0) {
17             throw new IllegalArgumentException();
18         }
19         this.next = start;
20     }
21
22     @Override
23     public boolean hasNext() {
24         return !done;
25     }
26
27     @Override

```

```

28     public Integer next() {
29         int value = next;
30         next = (next % 2 == 0) ? next / 2 : 3 * next + 1;
31         done = (value == 1);
32         return value;
33     }
34
35     @Override
36     public void remove() {
37         throw new UnsupportedOperationException();
38     }
39 }

```

```

1 CREATE FUNCTION javatest.seq(int)
2 RETURNS SETOF int
3 AS 'sandbox.PLJava.seq'
4 IMMUTABLE LANGUAGE java;

```

All things being equal it is better to create each result as needed. This usually reduces the memory footprint and avoids unnecessary work if the query has a LIMIT clause.

Single Tuples

A single tuple is returned in a `ResultSet`.

```

1 public static boolean singleWord(ResultSet receiver) throws SQLException {
2     receiver.updateString('English', 'hello');
3     receiver.updateString('Spanish', 'hola');
4     return true;
5 }

```

and

```

1 CREATE TYPE word AS (
2     English varchar,
3     Spanish varchar);
4
5 CREATE FUNCTION javatest.single_word()
6 RETURNS word
7 AS 'sandbox.PLJava.singleWord'
8 IMMUTABLE LANGUAGE java;

```

A valid result is indicated by returning `true`, a null result is indicated by returning `false`. A complex type can be passed into a java method in the same manner – it is a read-only `ResultSet` containing a single row.

Lists of Tuples

Returning lists of complex values requires a class implementing one of two interfaces.

`org.postgresql.pljava.ResultSetProvider`

A `ResultSetProvider` is used when the results can be created programmatically or on an as-needed basis.

```

01 public static ResultSetProvider listWords() {
02     return new WordProvider();
03 }
04
05 public static class WordProvider implements ResultSetProvider {
06     private final Map<String,String> words = new HashMap<String,String>();
07     private final Iterator<String> keys;
08
09     public WordProvider() {
10         words.put('one', 'uno');
11         words.put('two', 'dos');
12         words.put('three', 'tres');
13         words.put('four', 'quatro');
14         keys = words.keySet().iterator();
15     }
16
17     @Override
18     public boolean assignRowValues(ResultSet receiver, int currentRow)
19         throws SQLException {
20
21         if (!keys.hasNext()) {
22             return false;
23         }
24         String key = keys.next();
25         receiver.updateString('English', key);
26         receiver.updateString('Spanish', words.get(key));
27         return true;
28     }
29
30     @Override
31     public void close() throws SQLException {
32     }
33 }

```

and

```

1 CREATE FUNCTION javatest.list_words()
2 RETURNS SETOF word
3 AS 'sandbox.PLJava.listWords'
4 IMMUTABLE LANGUAGE java;

```

A `ResultSetHandle` is typically used when the method uses an internal query.

```

01 public static ResultSetHandle listUsers() {
02     return new UsersHandle();
03 }
04
05 public static class UsersHandle implements ResultSetHandle {
06     private Statement stmt;
07
08     @Override
09     public ResultSet getResultSet() throws SQLException {
10         stmt = DriverManager.getConnection('jdbc:default:connection').createStatement();
11         return stmt.executeQuery('SELECT * FROM pg_user');
12     }
13
14     @Override
15     public void close() throws SQLException {
16         stmt.close();
17     }
18 }

```

and

```

1 CREATE FUNCTION javatest.list_users()
2 RETURNS SETOF pg_user
3 AS 'sandbox.PLJava.listUsers'
4 LANGUAGE java;

```

The Interfaces

I have been unable to find a recent copy of the pljava jar in a standard maven repository. My solution was to extract the interfaces from the PL/Java source tarball. They are provided here for your convenience.

ResultSetProvider

```

01 // Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
02 // Distributed under the terms shown in the file COPYRIGHT
03 // found in the root folder of this project or at
04 // http://eng.tada.se/osprojects/COPYRIGHT.html
05
06 package org.postgresql.pljava;
07
08 import java.sql.ResultSet;
09 import java.sql.SQLException;
10
11
12 // An implementation of this interface is returned from functions and procedures
13 // that are declared to return <code>SET OF</code> a complex type. //Functions that
14 // return <code>SET OF</code> a simple type should simply return an
15 // {@link java.util.Iterator Iterator}.
16 // @author Thomas Hallgren
17
18 public interface ResultSetProvider
19 {
20
21     // This method is called once for each row that should be returned from
22     // a procedure that returns a set of rows. The receiver
23     // is a {@link org.postgresql.pljava.jdbc.SingleRowWriter SingleRowWriter}
24     // writer instance that is used for capturing the data for the row.
25     // @param receiver Receiver of values for the given row.
26     // @param currentRow Row number. First call will have row number 0.
27     // @return <code>true</code> if a new row was provided, <code>false</code>
28     // if not (end of data).
29     // @throws SQLException
30
31     boolean assignRowValues(ResultSet receiver, int currentRow)
32     throws SQLException;
33
34
35     // Called after the last row has returned or when the query evaluator decides
36     // that it does not need any more rows.
37     //
38     void close()
39     throws SQLException;
40 }

```

ResultSetHandle

```

01 // Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
02 // Distributed under the terms shown in the file COPYRIGHT
03 // found in the root directory of this distribution or at
04 // http://eng.tada.se/osprojects/COPYRIGHT.html
05
06 package org.postgresql.pljava;
07
08 import java.sql.ResultSet;
09 import java.sql.SQLException;
10
11
12 // An implementation of this interface is returned from functions and procedures
13 // that are declared to return <code>SET OF</code> a complex type in the form
14 // of a {@link java.sql.ResultSet}. The primary motivation for this interface is
15 // that an implementation that returns a ResultSet must be able to close the
16 // connection and statement when no more rows are requested.

```

```

17 // @author Thomas Hallgren
18
19 public interface ResultSetHandle
20 {
21
22     // An implementation of this method will probably execute a query
23     // and return the result of that query.
24     // @return The ResultSet that represents the rows to be returned.
25     // @throws SQLException
26
27     ResultSet getResultSet()
28     throws SQLException;
29
30
31     // Called after the last row has returned or when the query evaluator decides
32     // that it does not need any more rows.
33
34     void close()
35     throws SQLException;
36 }

```

Triggers

A database trigger is stored procedure that is automatically run during one of the three of the four CRUD (create-read-update-delete) operations.

- **insertion** - the trigger is provided the *new* value and is able to modify the values or prohibit the operation outright.
- **update** – the trigger is provided both *old* and *new* values. Again it is able to modify the values or prohibit the operation.
- **deletion** – the trigger is provided the *old* value. It is not able to modify the value but can prohibit the operation.

A trigger can be run before or after the operation. You would execute a trigger before an operation if you want to modify the values; you would execute it after an operation if you want to log the results.

Typical Usage

Insertion and Update: Data Validation

A pre-trigger on insert and update operations can be used to enforce data integrity and consistency. In this case the results are either accepted or the operation is prohibited.

Insertion and Update: Data Normalization and Sanitization

Sometimes values can have multiple representations or potentially be dangerous. A pre-trigger is a chance to clean up the data, e.g., to tidy up XML or replace < with < and > with >.

All Operations: Audit Logging

A post-trigger on all operations can be used to enforce audit logging. Applications can log their own actions but can't log direct access to the database. This is a solution to this problem.

A trigger can be run for each row or after completion of an entire statement. Update triggers can also be conditional.

Triggers can be used to create 'updateable views'.

PL/Java Implementation

Any java method can be used in a trigger provided it is a public static method returning void that takes a single argument, a *TriggerData* object. Triggers can be called "ON EACH ROW" or "ON STATEMENT".

TriggerDatas that are "ON EACH ROW" contain a single-row, read-only, ResultSet as the 'old' value on updates and deletions, and a single-row, updatable ResultSet as the 'new' value on insertions and updates. This can be used to modify content, log actions, etc.

```

01 public class AuditTrigger {
02
03     public static void auditFoobar(TriggerData td) throws SQLException {
04
05         Connection conn = DriverManager
06             .getConnection('jdbc:default:connection');
07         PreparedStatement ps = conn
08             .prepareStatement('insert into javatest.foobar_audit(what, whenn, data) values (?,
09             ?, ?::xml)');
10
11         if (td.isFiredByInsert()) {
12             ps.setString(1, 'INSERT');
13         } else if (td.isFiredByUpdate()) {
14             ps.setString(1, 'UPDATE');
15         } else if (td.isFiredByDelete()) {
16             ps.setString(1, 'DELETE');
17         }
18         ps.setTimestamp(2, new Timestamp(System.currentTimeMillis()));
19
20         ResultSet rs = td.getNew();
21         if (rs != null) {
22             ps.setString(3, toXml(rs));
23         } else {
24             ps.setNull(3, Types.VARCHAR);
25         }
26     }
27 }

```

```

24     }
25
26     ps.execute();
27     ps.close();
28 }
29
30 // simple marshaler. We could use JAXB or similar library
31 static String toXml(ResultSet rs) throws SQLException {
32     String foo = rs.getString(1);
33     if (rs.wasNull()) {
34         foo = '';
35     }
36     String bar = rs.getString(2);
37     if (rs.wasNull()) {
38         bar = '';
39     }
40     return String.format('<my-class><foo>%s</foo><bar>%s</bar></my-class>', foo, bar);
41 }
42 }

```

```

01 CREATE TABLE javatest.foobar (
02     foo   varchar(10),
03     bar   varchar(10)
04 );
05
06 CREATE TABLE javatest.foobar_audit (
07     what   varchar(10) not null,
08     whenn  timestamp not null,
09     data   xml
10 );
11
12 CREATE FUNCTION javatest.audit_foobar()
13 RETURNS trigger
14 AS 'sandbox.AuditTrigger.auditFoobar'
15 LANGUAGE 'java';
16
17 CREATE TRIGGER foobar_audit
18 AFTER INSERT OR UPDATE OR DELETE ON javatest.foobar
19 FOR EACH ROW
20 EXECUTE PROCEDURE javatest.audit_foobar();

```

Rules

A PostgreSQL extension is *Rules*. They are similar to triggers but a bit more flexible. One important difference is that Rules can be triggered on a SELECT statement, not just INSERT, UPDATE and DELETE.

Rules, unlike triggers, use standard functions.

The Interface

As before I have not been able to find a maven repository of a recent version and am including the files for your convenience.

TriggerData

```

01 // Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
02 // Distributed under the terms shown in the file COPYRIGHT
03 // found in the root folder of this project or at
04 // http://eng.tada.se/osprojects/COPYRIGHT.html
05
06 package org.postgresql.pljava;
07
08 import java.sql.ResultSet;
09 import java.sql.SQLException;
10
11
12 // The SQL 2003 spec. does not stipulate a standard way of mapping
13 // triggers to functions. The PLJava mapping use this interface. All
14 // functions that are intended to be triggers must be public, static,
15 // return void, and take a <code>TriggerData</code> as their argument.
16 //
17 // @author Thomas Hallgren
18
19 public interface TriggerData
20 {
21
22     // Returns the ResultSet that represents the new row. This ResultSet will
23     // be null for delete triggers and for triggers that was fired for
24     // statement.
25     // The returned set will be updateable and positioned on a
26     // valid row. When the trigger call returns, the trigger manager will se
27     // the changes that has been made to this row and construct a new tuple
28     // which will become the new or updated row.
29     //
30     // @return An updateable <code>ResultSet</code> containing one row or
31     // null
32     // @throws SQLException
33     // if the contained native buffer has gone stale.
34     //
35     ResultSet getNew() throws SQLException;
36
37
38     // Returns the ResultSet that represents the old row. This ResultSet will
39     // be null for insert triggers and for triggers that was fired for
40     // statement. The returned set will be read-only and positioned on a
41     // valid row.
42     //

```



```

43 // @return A read-only ResultSet containing one row or
44 // null.
45 // @throws SQLException
46 // if the contained native buffer has gone stale.
47 //
48 ResultSet getOld() throws SQLException;
49
50 //
51 // Returns the arguments for this trigger (as declared in the <code>CREATE TRIGGER</code> // E
52 // statement. If the trigger has no arguments, this method will return an
53 // array with size 0.
54 //
55 // @throws SQLException
56 // if the contained native buffer has gone stale.
57
58 String[] getArguments() throws SQLException;

```

```

01 // Returns the name of the trigger (as declared in the CREATE TRIGGER
02 // statement).
03 //
04 // @throws SQLException
05 // if the contained native buffer has gone stale.
06 //
07 String getName() throws SQLException;
08 /**
09 //Returns the name of the table for which this trigger was created (as
10 // declared in the <code>CREATE TRIGGER</code> statement). * * @throws SQLException* if the
11 contained native buffer has gone stale.
12 String getTableName() throws SQLException;
13 // Returns the name of the schema of the table for which this trigger was created (as * declared
14 in the <code>CREATE TRIGGER</code> statement).
15 // @throws SQLException * if the contained native buffer has gone stale. */
16 String getSchemaName() throws SQLException;
17 // Returns <code>true</code> if the trigger was fired after the statement or row action that it is
18 associated with.
19 // @throws SQLException * if the contained native buffer has gone stale.
20 boolean isFiredAfter() throws SQLException;
21 //Returns <code>true</code> if the trigger was fired before the * //statement or row action that it
22 is associated with. * * @throws SQLException * if //the contained native buffer has gone stale. */
23 boolean isFiredBefore() throws SQLException;
24 //Returns <code>true</code> if this trigger is fired once for each row * //(as opposed to once for
25 the entire statement). * * @throws SQLException * if the //contained native buffer has gone stale.
26 */
27 boolean isFiredForEachRow() throws SQLException;
28 //Returns <code>true</code> if this trigger is fired once for the entire //statement (as opposed to
29 once for each row). * * @throws SQLException * if the //contained native buffer has gone stale. */
30 boolean isFiredForStatement() throws SQLException;
31 //Returns <code>true</code> if this trigger was fired by a <code>DELETE</code>. * * @throws
32 SQLException * if the contained native //buffer has gone stale. */
33 boolean isFiredByDelete() throws SQLException;
34 //Returns <code>true</code> if this trigger was fired by an //<code>INSERT</code>. * * @throws
35 SQLException * if the contained native //buffer has gone stale. */
36 boolean isFiredByInsert() throws SQLException;
37 //Returns <code>true</code> if this trigger was fired by an //<code>UPDATE</code>. * * @throws
38 SQLException * if the contained native //buffer has gone stale. */
39 boolean isFiredByUpdate() throws SQLException;
40
41 // Returns the name of the table for which this trigger was created (as
42 // declared in the <code>CREATE TRIGGER</code> statement). * * @throws //SQLException* if the
43 contained native buffer has gone stale. */
44 String getTableName() throws SQLException;
45 // Returns the name of the schema of the table for which this trigger was created (as / declared in
46 the <code>CREATE TRIGGER</code> statement). * * @throws //SQLException * if the contained native
47 buffer has gone stale. */
48 String getSchemaName() throws SQLException;
49 //Returns <code>true</code> if the trigger was fired after the statement // or row action that it
50 is associated with. * * @throws SQLException * if the //contained native buffer has gone stale. */
51 boolean isFiredAfter() throws SQLException;
52 // Returns <code>true</code> if the trigger was fired before the * //statement or row action that
53 it is associated with. * * @throws SQLException * if //the contained native buffer has gone stale.
54 */
55 boolean isFiredBefore() throws SQLException;
56 // Returns <code>true</code> if this trigger is fired once for each row * //(as opposed to once for
57 the entire statement). * * @throws SQLException * if the //contained native buffer has gone stale.
58 */
59 boolean isFiredForEachRow() throws SQLException;
60 // Returns <code>true</code> if this trigger is fired once for the entire // statement (as opposed
61 to once for each row). * * @throws SQLException * if the //contained native buffer has gone stale.
62 */
63 boolean isFiredForStatement() throws SQLException;
64 // Returns <code>true</code> if this trigger was fired by a //<code>DELETE</code>. * * @throws
65 SQLException * if the contained native //buffer has gone stale. */
66 boolean isFiredByDelete() throws SQLException;
67 // Returns <code>true</code> if this trigger was fired by an //<code>INSERT</code>. * * @throws
68 SQLException * if the contained native //buffer has gone stale. */
69 boolean isFiredByInsert() throws SQLException;
70 // Returns <code>true</code> if this trigger was fired by an //<code>UPDATE</code>. * * @throws
71 SQLException * if the contained native //buffer has gone stale. */
72 boolean isFiredByUpdate() throws SQLException; }/**
73 // Returns the name of the table for which this trigger was created (as
74 // declared in the <code>CREATE TRIGGER</code> statement). * * @throws //SQLException* if the
75 contained native buffer has gone stale. */
76 String getTableName() throws SQLException;
77 // Returns the name of the schema of the table for which this trigger was created (as // declared
78 in the <code>CREATE TRIGGER</code> statement). * * @throws //SQLException * if the contained native
79 buffer has gone stale. */
80 String getSchemaName() throws SQLException;
81 // Returns <code>true</code> if the trigger was fired after the //statement * or row action that
82 it is associated with. * * @throws SQLException * if //the contained native buffer has gone stale.
83 */
84 boolean isFiredAfter() throws SQLException;

```



```

59 // Returns <code>true</code> if the trigger was fired before the * //statement or row action that
   // it is associated with. * * @throws SQLException * if //the contained native buffer has gone stale.
   */
60 boolean isFiredBefore() throws SQLException;
61 // Returns <code>true</code> if this trigger is fired once for each row * (//as opposed to once for
   // the entire statement). * * @throws SQLException * if the //contained native buffer has gone stale.
   */
62 boolean isFiredForEachRow() throws SQLException;
63 // Returns <code>true</code> if this trigger is fired once for the entire // statement (as opposed
   // to once for each row). * * @throws SQLException * if the //contained native buffer has gone stale.
   */
64 boolean isFiredForStatement() throws SQLException;
65 // Returns <code>true</code> if this trigger was fired by a //<code>DELETE</code>. * * @throws
   // SQLException * if the contained native //buffer has gone stale. */
66 boolean isFiredByDelete() throws SQLException;
67 // Returns <code>true</code> if this trigger was fired by an //<code>INSERT</code>. * * @throws
   // SQLException * if the contained native //buffer has gone stale. */
68 boolean isFiredByInsert() throws SQLException;
69 // Returns <code>true</code> if this trigger was fired by an //<code>UPDATE</code>. * * @throws
   // SQLException * if the contained native //buffer has gone stale. */
70 boolean isFiredByUpdate() throws SQLException; }

```

TriggerException

```

01 // Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
02 // Distributed under the terms shown in the file COPYRIGHT
03 // found in the root folder of this project or at
04 // http://eng.tada.se/osprojects/COPYRIGHT.html
05
06 package org.postgresql.pljava;
07
08 import java.sql.SQLException;
09
10
11 // An exception specially suited to be thrown from within a method
12 // designated to be a trigger function. The message generated by
13 // this exception will contain information on what trigger and
14 // what relation it was that caused the exception
15 //
16 // @author Thomas Hallgren
17
18 public class TriggerException extends SQLException
19 {
20     private static final long serialVersionUID = 5543711707414329116L;
21
22     private static boolean s_recursionLock = false;
23
24     public static final String TRIGGER_ACTION_EXCEPTION = '09000';
25
26     private static final String makeMessage(TriggerData td, String message)
27     {
28         StringBuffer bld = new StringBuffer();
29         bld.append('In Trigger ');
30         if(!s_recursionLock)
31         {
32             s_recursionLock = true;
33             try
34             {
35                 bld.append(td.getName());
36                 bld.append(' on relation ');
37                 bld.append(td.getTableName());
38             }
39             catch(SQLException e)
40             {
41                 bld.append('(exception while generating exception message)');
42             }
43             finally
44             {
45                 s_recursionLock = false;
46             }
47         }
48         if(message != null)
49         {
50             bld.append(': ');
51             bld.append(message);
52         }
53         return bld.toString();
54     }
55
56
57     // Create an exception based on the <code>TriggerData</code> that was
58     // passed to the trigger method.
59     // @param td The <code>TriggerData</code> that was passed to the trigger
60     // method.
61
62     public TriggerException(TriggerData td)
63     {
64         super(makeMessage(td, null), TRIGGER_ACTION_EXCEPTION);
65     }
66
67
68     // Create an exception based on the <code>TriggerData</code> that was
69     // passed to the trigger method and an additional message.
70     // @param td The <code>TriggerData</code> that was passed to the trigger
71     // method.
72     // @param reason An additional message with info about the exception.
73
74     public TriggerException(TriggerData td, String reason)
75     {
76         super(makeMessage(td, reason), TRIGGER_ACTION_EXCEPTION);
77     }
78 }

```

User-defined types in the database are controversial. They're not standard – at some point the DBA has to create them – and this introduces portability issues. Standard tools won't know about them. You must access them via the 'struct' methods in ResultSets and PreparedStatements.

On the other hand there are a LOT of things that are otherwise only supported as byte[]. This prevents database functions and stored procedures from easily manipulating them.

What would be a good user-defined type? It must be atomic and it must be possible to do meaningful work via stored procedures. N.B., a database user-defined type is not the same thing as a java class. Nearly all java classes should be stored as standard tuples and you should only use database UDTs if there's a compelling reason.

A touchstone I like is asking whether you're ever tempted to cache immutable information about the type, vs. about the tuple, in addition to the object itself. E.g., a X.509 digital certificate has a number of immutable fields that would be valid search terms but it's expensive to extract that information for every row. (Sidenote: you can use triggers to extract the information when the record is inserted and updated. This ensures the cached values are always accurate.)

Examples:

- complex numbers (stored procedures: arithmetic)
- rational numbers (stored procedures: arithmetic)
- galois field numbers (stored procedures: arithmetic modulo a fixed value)
- images (stored procedures: get dimensions)
- PDF documents (stored procedures: extract elements)
- digital certificates and private keys (stored procedures: crypto)

Something that should also be addressed is the proper language for implementation. It's easy to prototype in PL/Java but you can make a strong argument that *types* should be ultimately implemented as a standard PostgreSQL extensions since they're more likely to be available in the future when you're looking at a 20-year-old dump. In some important ways this is just a small part of the problem – the issue isn't whether the actual storage and function implementation is written in C or java, it's how it's tied into the rest of the system.

PL/Java Implementation

A PL/Java user defined type must implement the *java.sql.SQLData* interface, a static method that creates the object from a String, and an instance method that creates a String from the object. These methods must be complementary – it must be possible to run a value through a full cycle in either direction and get the original value back.

N.B., this is often impossible with doubles – this is why you get numbers like 4.000000001 or 2.999999999. In these cases you have to do the best you can and warn the user.

In many cases an object can be stored more efficiently in a binary format. In PostgreSQL terms these are TOAST types. This is handled by implementing two new methods that work with SQLInput and SQLOutput streams.

A simple implementation of a rational type follows.

```
01 public class Rational implements SQLData {
02     private long numerator;
03     private long denominator;
04     private String typeName;
05
06     public static Rational parse(String input, String typeName)
07         throws SQLException {
08         Pattern pattern = Pattern.compile('(-?[0-9]+)( /* (-?[0-9]+)?)?');
09         Matcher matcher = pattern.matcher(input);
10         if (!matcher.matches()) {
11             throw new SQLException("Unable to parse rational from string \"" + input
12                                     + "\"");
13         }
14         if (matcher.groupCount() == 3) {
15             if (matcher.group(3) == null) {
16                 return new Rational(Long.parseLong(matcher.group(1)));
17             }
18             return new Rational(Long.parseLong(matcher.group(1)),
19                                 Long.parseLong(matcher.group(3)));
20         }
21         throw new SQLException("invalid format: \"" + input
22                                 + "\"");
23     }
24
25     public Rational(long numerator) throws SQLException {
26         this(numerator, 1);
27     }
28
29     public Rational(long numerator, long denominator) throws SQLException {
30         if (denominator == 0) {
31             throw new SQLException("denominator must be non-zero");
32         }
33
34         // do a little bit of normalization
35         if (denominator < 0) {
36             numerator = -numerator;
37             denominator = -denominator;
38         }
39
40         this.numerator = numerator;
41         this.denominator = denominator;
```

```

42     }
43
44     public Rational(int numerator, int denominator, String typeName)
45         throws SQLException {
46         this(numerator, denominator);
47         this.typeName = typeName;
48     }
49
50     public String getSQLTypeName() {
51         return typeName;
52     }
53
54     public void readSQL(SQLInput stream, String typeName) throws SQLException {
55         this.numerator = stream.readLong();
56         this.denominator = stream.readLong();
57         this.typeName = typeName;
58     }
59
60     public void writeSQL(SQLOutput stream) throws SQLException {
61         stream.writeLong(numerator);
62         stream.writeLong(denominator);
63     }
64
65     public String toString() {
66         String value = null;
67         if (denominator == 1) {
68             value = String.valueOf(numerator);
69         } else {
70             value = String.format('%d/%d', numerator, denominator);
71         }
72         return value;
73     }
74
75     /*
76     * Meaningful code that actually does something with this type was
77     * intentionally left out.
78     */
79 }

```

and

```

01 /* The shell type */
02 CREATE TYPE javatest.rational;
03
04 /* The scalar input function */
05 CREATE FUNCTION javatest.rational_in(cstring)
06     RETURNS javatest.rational
07     AS 'UDT[sandbox.Rational] input'
08     LANGUAGE java IMMUTABLE STRICT;
09
10 /* The scalar output function */
11 CREATE FUNCTION javatest.rational_out(javatest.rational)
12     RETURNS cstring
13     AS 'UDT[sandbox.Rational] output'
14     LANGUAGE java IMMUTABLE STRICT;
15
16 /* The scalar receive function */
17 CREATE FUNCTION javatest.rational_recv(internal)
18     RETURNS javatest.rational
19     AS 'UDT[sandbox.Rational] receive'
20     LANGUAGE java IMMUTABLE STRICT;
21
22 /* The scalar send function */
23 CREATE FUNCTION javatest.rational_send(javatest.rational)
24     RETURNS bytea
25     AS 'UDT[sandbox.Rational] send'
26     LANGUAGE java IMMUTABLE STRICT;
27
28 CREATE TYPE javatest.rational (
29     internallength = 16,
30     input = javatest.rational_in,
31     output = javatest.rational_out,
32     receive = javatest.rational_recv,
33     send = javatest.rational_send,
34     alignment = int);

```

Type modifiers

PostgreSQL allows types to have modifiers. Examples are in 'varchar(200)' or 'numeric(8,2)'.

PL/Java does not currently support this functionality (via the 'typmod_in' and 'typmod_out' methods) but I have submitted a request for it.

Casts

Custom types aren't particularly useful if all you can do is store and retrieve the values as opaque objects. Why not use bytea and be done with it?

In fact there are many UDTs where it makes sense to be able to cast a UDT to a different type. Numeric types, like complex or rational numbers, should be able to be converted to and from the standard integer and floating number types (albeit with limitations).

This should be done with restraint.

Casts are implemented as single argument static methods. In the java world these methods are often named *newInstance* so I'm doing

the same here.

```
01 public static Rational newInstance(String input) throws SQLException {
02     if (input == null) {
03         return null;
04     }
05     return parse(input, 'javatest.rational');
06 }
07
08 public static Rational newInstance(int value) throws SQLException {
09     return new Rational(value);
10 }
11
12 public static Rational newInstance(Integer value) throws SQLException {
13     if (value == null) {
14         return null;
15     }
16     return new Rational(value.intValue());
17 }
18
19 public static Rational newInstance(long value) throws SQLException {
20     return new Rational(value);
21 }
22
23 public static Rational newInstance(Long value) throws SQLException {
24     if (value == null) {
25         return null;
26     }
27     return new Rational(value.longValue());
28 }
29
30 public static Double value(Rational value) throws SQLException {
31     if (value == null) {
32         return null;
33     }
34     return value.doubleValue();
35 }
```

and

```
01 CREATE FUNCTION javatest.rational_string_as_rational(varchar) RETURNS javatest.rational
02     AS 'sandbox.Rational.newInstance'
03     LANGUAGE JAVA IMMUTABLE STRICT;
04
05 CREATE FUNCTION javatest.rational_int_as_rational(int4) RETURNS javatest.rational
06     AS 'sandbox.Rational.newInstance'
07     LANGUAGE JAVA IMMUTABLE STRICT;
08
09 CREATE FUNCTION javatest.rational_long_as_rational(int8) RETURNS javatest.rational
10     AS 'sandbox.Rational.newInstance'
11     LANGUAGE JAVA IMMUTABLE STRICT;
12
13 CREATE FUNCTION javatest.rational_as_double(javatest.rational) RETURNS float8
14     AS 'sandbox.Rational.value'
15     LANGUAGE JAVA IMMUTABLE STRICT;
16
17 CREATE CAST (varchar AS javatest.rational)
18     WITH FUNCTION javatest.rational_string_as_rational(varchar)
19     AS ASSIGNMENT;
20
21 CREATE CAST (int4 AS javatest.rational)
22     WITH FUNCTION javatest.rational_int_as_rational(int4)
23     AS ASSIGNMENT;
24
25 CREATE CAST (int8 AS javatest.rational)
26     WITH FUNCTION javatest.rational_long_as_rational(int8)
27     AS ASSIGNMENT;
28
29 CREATE CAST (javatest.rational AS float8)
30     WITH FUNCTION javatest.rational_as_double(javatest.rational)
31     AS ASSIGNMENT;
```

(Sidenote: *STRICT* means that the function will return NULL if any argument is NULL. This allows the database to make some optimizations.)

(Sidenote: we may only be able to use the *IMMUTABLE* flag if the java objects are also immutable. We should probably make our Rational objects immutable since the other numeric types are immutable.)

Aggregate Functions

What about *min()*? Rational numbers are a numeric type so shouldn't they support all of the standard aggregate functions?

Defining new aggregate functions is straightforward. Simple aggregate functions only need a static member function that take two UDT values and return one. This is easy to see with maximums, minimums, sums, products, etc. More complex aggregates require an ancillary UDT that contains state information, a static method that takes one state UDT and one UDT and returns a state UDT, and a finalization method that takes the final state UDT and produces the results. This is easy to see with averages – you need a state type that contains a counter and a running sum.

Several examples of the former type of aggregate function follow.

```
01 // compare two Rational objects. We use BigInteger to avoid overflow.
02 public static int compare(Rational p, Rational q) {
03     if (p == null) {
04         return 1;
```

```

05     } else if (q == null) {
06         return -1;
07     }
08     BigInteger l =
09     BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator()));
10     BigInteger r =
11     BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator()));
12     return l.compareTo(r);
13 }
14 public static Rational min(Rational p, Rational q) {
15     if ((p == null) || (q == null)) {
16         return null;
17     }
18     return (p.compareTo(q) <= 0) ? p : q;
19 }
20 public static Rational max(Rational p, Rational q) {
21     if ((p == null) || (q == null)) {
22         return null;
23     }
24     return (q.compareTo(p) < 0) ? p : q;
25 }
26
27 public static Rational add(Rational p, Rational q) throws SQLException {
28     if ((p == null) || (q == null)) {
29         return null;
30     }
31     BigInteger n =
32     BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator())).add(
33     BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator())));
34     BigInteger d =
35     BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
36     BigInteger gcd = n.gcd(d);
37     n = n.divide(gcd);
38     d = d.divide(gcd);
39     return new Rational(n.longValue(), d.longValue());
40 }

```

and

```

01 CREATE FUNCTION javatest.min(javatest.rational, javatest.rational) RETURNS javatest.rational
02 AS 'sandbox.Rational.min'
03 LANGUAGE JAVA IMMUTABLE STRICT;
04
05 CREATE FUNCTION javatest.max(javatest.rational, javatest.rational) RETURNS javatest.rational
06 AS 'sandbox.Rational.max'
07 LANGUAGE JAVA IMMUTABLE STRICT;
08
09 CREATE AGGREGATE min(javatest.rational) (
10     sfunc = javatest.min,
11     stype = javatest.rational
12 );
13
14 CREATE AGGREGATE max(javatest.rational) (
15     sfunc = javatest.max,
16     stype = javatest.rational
17 );
18
19 CREATE AGGREGATE sum(javatest.rational) (
20     sfunc = javatest.add,
21     stype = javatest.rational
22 );

```

Integration with Hibernate

It is possible to link PL/Java user-defined types and Hibernate user-defined types. Warning: the hibernate code is highly database-specific.

This is the hibernate user-defined type. PostgreSQL 9.1 does not support the STRUCT type and uses strings instead. We don't have to use the PL/Java user-defined data type to perform the marshaling but it ensures consistency. *TheDbRationalType* is the *Rational* class above. The same class could be used in both places but would introduce dependency on a Hibernate interface into the PL/Java class. This may be acceptable if you extract that single interface from the Hibernate source code.

```

001 public class Rational implements UserType, Serializable {
002     private final int[] sqlTypesSupported = new int[] { Types.OTHER };
003     private long numerator;
004     private long denominator;
005
006     public Rational() {
007         numerator = 0;
008         denominator = 1;
009     }
010
011     public Rational(long numerator, long denominator) {
012         this.numerator = numerator;
013         this.denominator = denominator;
014     }
015
016     public long getNumerator() {
017         return numerator;
018     }
019
020     public long getDenominator() {
021         return denominator;
022     }
023 }

```

```

024 @Override
025 public Object assemble(Serializable cached, Object owner)
026     throws HibernateException {
027     if (!(cached instanceof Rational)) {
028         throw new HibernateException('invalid argument');
029     }
030     Rational r = (Rational) cached;
031     return new Rational(r.getNumerator(), r.getDenominator());
032 }
033
034 @Override
035 public Serializable disassemble(Object value) throws HibernateException {
036     if (!(value instanceof Rational)) {
037         throw new HibernateException('invalid argument');
038     }
039     return (Rational) value;
040 }
041
042 @Override
043 public Object deepCopy(Object value) throws HibernateException {
044     if (value == null) {
045         return null
046     }
047     if (!(value instanceof Rational)) {
048         throw new HibernateException('invalid argument');
049     }
050     Rational v = (Rational) value;
051     return new Rational(v.getNumerator(), v.getDenominator());
052 }
053
054 @Override
055 public boolean isMutable() {
056     return true;
057 }
058
059 //
060 // important: PGObject is postgresql-specific
061 //
062 @Override
063 public Object nullSafeGet(ResultSet rs, String[] names, Object owners)
064     throws HibernateException, SQLException {
065     PGObject pgo = (PGObject) rs.getObject(names[0]);
066     if (rs.wasNull()) {
067         return null;
068     }
069     TheDbRationalType r = TheDbRationalType.parse(pgo.getValue(), 'rational');
070     return new Rational(r.getNumerator(), r.getDenominator());
071 }
072
073 //
074 // important: using Types.OTHER may be postgresql-specific
075 //
076 @Override
077 public void nullSafeSet(PreparedStatement ps, Object value, int index)
078     throws HibernateException, SQLException {
079     if (value == null) {
080         ps.setNull(index, Types.OTHER);
081     } else if (!(value instanceof Rational)) {
082         throw new HibernateException('invalid argument');
083     } else {
084         Rational t = (Rational) value;
085         ps.setObject(index,
086             new TheDbRationalType(t.getNumerator(), t.getDenominator()), Types.OTHER);
087     }
088 }
089
090 @Override
091 public Object replace(Object original, Object target, Object owner)
092     throws HibernateException {
093     if (!(original instanceof Rational)
094         || !(target instanceof Rational)) {
095         throw new HibernateException('invalid argument');
096     }
097     Rational r = (Rational) original;
098     return new Rational(r.getNumerator(), r.getDenominator());
099 }
100
101 @Override
102 public Class returnedClass() {
103     return Rational.class;
104 }
105
106 @Override
107 public int[] sqlTypes() {
108     return sqlTypesSupported;
109 }
110
111 @Override
112 public String toString() {
113     String value = '';
114     if (denominator == 1) {
115         value = String.valueOf(numerator);
116     } else {
117         value = String.format('%d/%d', numerator, denominator);
118     }
119     return value;
120 }
121
122 // for UserType
123 @Override
124 public int hashCode(Object value) {
125     Rational r = (Rational) value;
126     return (int) (31 * r.getNumerator() + r.getDenominator());

```

```

127     }
128
129     @Override
130     public int hashCode() {
131         return hashCode(this);
132     }
133
134     // for UserType
135     @Override
136     public boolean equals(Object left, Object right) {
137         if (left == right) {
138             return true;
139         }
140         if ((left == null) || (right == null)) {
141             return false;
142         }
143         if (!(left instanceof Rational) || !(right instanceof Rational)) {
144             return false;
145         }
146
147         Rational l = (Rational) left;
148         Rational r = (Rational) right;
149         return (l.getNumerator() == r.getNumerator())
150             && (l.getDenominator() == r.getDenominator());
151     }
152
153     @Override
154     public boolean equals(Object value) {
155         return equals(this, value);
156     }
157 }

```

CustomTypes.hbm.xml

```

01 <?xml version='1.0' encoding='utf-8'?>
02 <!DOCTYPE hibernate-mapping PUBLIC
03     '-//Hibernate/Hibernate Mapping DTD 3.0//EN'
04     'http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd '>
05
06 <hibernate-mapping>
07
08     <typedef name='javatest.rational' class='sandbox.RationalType'/>
09
10 </hibernate-mapping>

```

TestTable.hbm.xml

```

01 <?xml version='1.0' encoding='utf-8'?>
02 <!DOCTYPE hibernate-mapping PUBLIC
03     '-//Hibernate/Hibernate Mapping DTD 3.0//EN'
04     'http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd '>
05
06 <hibernate-mapping>
07
08     <class name='sandbox.TestTable' table='test_table'>
09         <id name='id' />
10         <property name='value' type='javatest.rational' />
11     </class>
12
13 </hibernate-mapping>

```

Operators

Operators are normal PL/Java methods that are also marked as operators via the CREATE OPERATOR statement.

Basic arithmetic for rational numbers is supported as

```

01 public static Rational negate(Rational p) throws SQLException {
02     if (p == null) {
03         return null;
04     }
05     return new Rational(-p.getNumerator(), p.getDenominator());
06 }
07
08 public static Rational add(Rational p, Rational q) throws SQLException {
09     if ((p == null) || (q == null)) {
10         return null;
11     }
12     BigInteger n =
13         BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator())).add(
14             BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator())));
15     BigInteger d =
16         BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
17     BigInteger gcd = n.gcd(d);
18     n = n.divide(gcd);
19     d = d.divide(gcd);
20     return new Rational(n.longValue(), d.longValue());
21 }
22
23 public static Rational subtract(Rational p, Rational q) throws SQLException {
24     if ((p == null) || (q == null)) {
25         return null;
26     }
27     BigInteger n =
28         BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator())).subtract(
29             BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator())));
30     BigInteger d =
31         BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
32     BigInteger gcd = n.gcd(d);
33     n = n.divide(gcd);
34     d = d.divide(gcd);
35     return new Rational(n.longValue(), d.longValue());
36 }

```



```

30     d = d.divide(gcd);
31     return new Rational(n.longValue(), d.longValue());
32 }
33
34 public static Rational multiply(Rational p, Rational q) throws SQLException {
35     if ((p == null) || (q == null)) {
36         return null;
37     }
38     BigInteger n =
39     BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getNumerator()));
40     BigInteger d =
41     BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
42     BigInteger gcd = n.gcd(d);
43     n = n.divide(gcd);
44     d = d.divide(gcd);
45     return new Rational(n.longValue(), d.longValue());
46 }

```

and

```

01 CREATE FUNCTION javatest.rational_negate(javatest.rational) RETURNS javatest.rational
02 AS 'sandbox.Rational.negate'
03 LANGUAGE JAVA IMMUTABLE STRICT;
04
05 CREATE FUNCTION javatest.rational_add(javatest.rational, javatest.rational)
06 RETURNS javatest.rational
07 AS 'sandbox.Rational.add'
08 LANGUAGE JAVA IMMUTABLE STRICT;
09
10 CREATE FUNCTION javatest.rational_subtract(javatest.rational, javatest.rational)
11 RETURNS javatest.rational
12 AS 'sandbox.Rational.subtract'
13 LANGUAGE JAVA IMMUTABLE STRICT;
14
15 CREATE FUNCTION javatest.rational_multiply(javatest.rational, javatest.rational)
16 RETURNS javatest.rational
17 AS 'sandbox.Rational.multiply'
18 LANGUAGE JAVA IMMUTABLE STRICT;
19
20 CREATE FUNCTION javatest.rational_divide(javatest.rational, javatest.rational)
21 RETURNS javatest.rational
22 AS 'sandbox.Rational.divide'
23 LANGUAGE JAVA IMMUTABLE STRICT;
24
25 CREATE OPERATOR - (
26     rightarg = javatest.rational, procedure.rational_negate
27 );
28
29 CREATE OPERATOR + (
30     leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_add,
31     commutator = +
32 );
33
34 CREATE OPERATOR - (
35     leftarg = javatest.rational, rightarg = javatest.rational, procedure =
36     javatest.rational_subtract
37 );
38
39 CREATE OPERATOR * (
40     leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_multiply,
41     commutator = *
42 );
43
44 CREATE OPERATOR / (
45     leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_divide
46 );

```

The operator characters are one to 63 characters from the set “+ – * / < > = ~ ! @ # % ^ & | ` ?” with a few restrictions to avoid confusion with the start of SQL comments.

The *commutator* operator is a second operator (possibly the same) that has the same results if the left and right values are swapped. This is used by the optimizer.

The *negator* operator is one that the opposite results if the left and right values are swapped. It is only valid on procedures that return a boolean value. Again this is used by the optimizer.

Ordering Operators

Many UDTs can be ordered in some manner. This may be something obvious, e.g., ordering rational numbers, or something a bit more arbitrary, e.g., ordering complex numbers.

We can define ordering operations in the same manner as above. N.B., there is no longer anything special about these operators – with an unfamiliar UDT you can’t assume that < really means “less than”. The sole exception is “!=” which is always rewritten as “≠” by the parser.

```

01 public static int compare(Rational p, Rational q) {
02     if (p == null) {
03         return 1;
04     } else if (q == null) {
05         return -1;
06     }
07     BigInteger l =
08     BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator()));
09     BigInteger r =

```

```

009 BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator()));
10     return l.compareTo(r);
11 }
12 public int compareTo(Rational p) {
13     return compare(this, p);
14 }
15
16 public static int compare(Rational p, double q) {
17     if (p == null) {
18         return 1;
19     }
20     double d = p.doubleValue();
21     return (d < q) ? -1 : ((d == q) ? 0 : 1);
22 }
23
24 public int compareTo(double q) {
25     return compare(this, q);
26 }
27
28 public static boolean lessThan(Rational p, Rational q) {
29     return compare(p, q) < 0;
30 }
31
32 public static boolean lessThanOrEquals(Rational p, Rational q) {
33     return compare(p, q) <= 0;
34 }
35
36 public static boolean equals(Rational p, Rational q) {
37     return compare(p, q) == 0;
38 }
39
40 public static boolean greaterThan(Rational p, Rational q) {
41     return compare(p, q) > 0;
42 }
43
44 public static boolean lessThan(Rational p, double q) {
45     if (p == null) {
46         return false;
47     }
48     return p.compareTo(q) < 0;
49 }
50
51 public static boolean lessThanOrEquals(Rational p, double q) {
52     if (p == null) {
53         return false;
54     }
55     return p.compareTo(q) == 0;
56 }
57
58 public static boolean greaterThan(Rational p, double q) {
59     if (p == null) {
60         return true;
61     }
62     return p.compareTo(q) > 0;
63 }

```

Note that I've defined methods to compare either two rational numbers or one rational number and one double number.

```

001 CREATE FUNCTION javatest.rational_lt(javatest.rational, javatest.rational)
002     RETURNS bool
003     AS 'sandbox.Rational.lessThan'
004     LANGUAGE JAVA IMMUTABLE STRICT;
005
006 CREATE FUNCTION javatest.rational_le(javatest.rational, javatest.rational)
007     RETURNS bool
008     AS 'sandbox.Rational.lessThanOrEquals'
009     LANGUAGE JAVA IMMUTABLE STRICT;
010
011 CREATE FUNCTION javatest.rational_eq(javatest.rational, javatest.rational)
012     RETURNS bool
013     AS 'sandbox.Rational.equals'
014     LANGUAGE JAVA IMMUTABLE STRICT;
015
016 CREATE FUNCTION javatest.rational_ge(javatest.rational, javatest.rational)
017     RETURNS bool
018     AS 'sandbox.Rational.greaterThanOrEquals'
019     LANGUAGE JAVA IMMUTABLE STRICT;
020
021 CREATE FUNCTION javatest.rational_gt(javatest.rational, javatest.rational)
022     RETURNS bool
023     AS 'sandbox.Rational.greaterThan'
024     LANGUAGE JAVA IMMUTABLE STRICT;
025
026 CREATE FUNCTION javatest.rational_cmp(javatest.rational, javatest.rational)
027     RETURNS int
028     AS 'sandbox.Rational.compare'
029     LANGUAGE JAVA IMMUTABLE STRICT;
030
031 CREATE FUNCTION javatest.rational_lt(javatest.rational, float8)
032     RETURNS bool
033     AS 'sandbox.Rational.lessThan'
034     LANGUAGE JAVA IMMUTABLE STRICT;
035
036 CREATE FUNCTION javatest.rational_le(javatest.rational, float8)
037     RETURNS bool
038     AS 'sandbox.Rational.lessThanOrEquals'
039     LANGUAGE JAVA IMMUTABLE STRICT;
040
041 CREATE FUNCTION javatest.rational_eq(javatest.rational, float8)
042     RETURNS bool
043     AS 'sandbox.Rational.equals'

```

```

044 LANGUAGE JAVA IMMUTABLE STRICT;
045
046 CREATE FUNCTION javatest.rational_ge(javatest.rational, float8)
047 RETURNS bool
048 AS 'sandbox.Rational.greaterThanOrEquals'
049 LANGUAGE JAVA IMMUTABLE STRICT;
050
051 CREATE FUNCTION javatest.rational_gt(javatest.rational, float8)
052 RETURNS bool
053 AS 'sandbox.Rational.greaterThan'
054 LANGUAGE JAVA IMMUTABLE STRICT;
055
056 CREATE OPERATOR < (
057 leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_lt,
058 commutator = >, negator = >=,
059 restrict = scalarlttsel, join = scalarlttjoinssel, merges
060 );
061
062 CREATE OPERATOR <= (
063 leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_le,
064 commutator = >=, negator = >,
065 restrict = scalarlttsel, join = scalarlttjoinssel, merges
066 );
067
068 CREATE OPERATOR = (
069 leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_eq,
070 commutator = =, negator = <>, hashes, merges
071 );
072
073 CREATE OPERATOR >= (
074 leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_lt,
075 commutator = <=, negator = <,
076 restrict = scalarlttsel, join = scalarlttjoinssel, merges
077 );
078
079 CREATE OPERATOR > (
080 leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_le,
081 commutator = <=, negator = <,
082 restrict = scalargtsel, join = scalargttjoinssel, merges
083 );
084
085 CREATE OPERATOR < (
086 leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_lt,
087 commutator = >, negator = >=
088 );
089
090 CREATE OPERATOR <= (
091 leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_le,
092 commutator = >=, negator = >
093 );
094
095 CREATE OPERATOR = (
096 leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_eq,
097 commutator = =, negator = <>
098 );
099
100 CREATE OPERATOR >= (
101 leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_ge,
102 commutator = <=, negator = <
103 );
104
105 CREATE OPERATOR > (
106 leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_gt,
107 commutator = <, negator = <=
108 );

```

Restrict is an optimization estimator procedure. It's usually safe to use the appropriate standard procedure.

Join is an optimization estimator procedure. It's usually safe to use the appropriate standard procedure.

Hashes indicates that the operator can be used in hash joins.

Merges indicates that the operator can be used in merge joins.

Indexes

Indexes are used in three places – to enforce uniqueness constraints and to speed up WHERE and JOIN clauses.

```

01 -- btree join
02 CREATE OPERATOR CLASS rational_ops
03 DEFAULT FOR TYPE javatest.rational USING btree AS
04 OPERATOR 1 <,
05 OPERATOR 2 <=,
06 OPERATOR 3 =,
07 OPERATOR 4 >=,
08 OPERATOR 5 >,
09 FUNCTION 1 javatest.rational_cmp(javatest.rational, javatest.rational);
10
11 -- hash join
12 CREATE OPERATOR CLASS rational_ops
13 DEFAULT FOR TYPE javatest.rational USING hash AS
14 OPERATOR 1 =,
15 FUNCTION 1 javatest.rational_hashCode(javatest.rational);

```

Operator Families

Finally, PostgreSQL has the concept of “Operator Families” that group related operator classes under a single umbrella. For instance you might have one family that supports cross-comparison between int2, int4 and int8 values. Each can be specified individually but by creating an operator family you give a few more hints to the PostgreSQL optimizer.

More Information

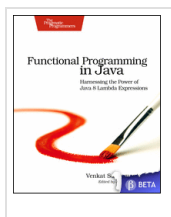
- [CREATE TYPE \(PostgreSQL\)](#)
- [PostgreSQL 'create trigger' documentation](#)
- [PostgreSQL 'create rule' documentation](#)
- [CREATE OPERATOR \(PostgreSQL\)](#)
- [CREATE OPERATOR CLASS \(PostgreSQL\)](#)
- [CREATE OPERATOR FAMILY \(PostgreSQL\)](#)
- [Operator Optimization \(PostgreSQL\)](#)
- [Interfacing Extensions To Indexes \(PostgreSQL\)](#)
- [Creating a Scalar UDT in Java \(user guide\)](#)
- [CREATE AGGREGATE documentation \(PostgreSQL\)](#)
- [CREATE CAST documentation \(PostgreSQL\)](#)
- [CREATE TYPE documentation \(PostgreSQL\)](#)
- [CREATE OPERATOR documentation \(PostgreSQL\)](#)
- [CREATE OPERATOR CLASS documentation \(PostgreSQL\)](#)
- [Interfacing user-defined types to indexes \(PostgreSQL\)](#)

Reference: [Introduction To PostgreSQL PL/Java, Part 1](#) , [Introduction To PostgreSQL PL/Java, Part 2: Working With Lists](#) , [Introduction To PostgreSQL PL/Java, Part 3: Triggers](#) , [Introduction To PostgreSQL PL/Java, Part 4: User Defined Types](#) , [Introduction To PostgreSQL/PLJava, Part 5: Operations And Indexes](#) from our JCG partner Bear Giles at the [Invariant Properties](#) blog.

You might also like:

- [Why PostgreSQL is so Awesome](#)
- [On Java 8's introduction of Optional](#)
- [Java EE 6 Testing Part II – Introduction to Arquillian and ShrinkWrap](#)

Related Whitepaper:



Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions

Get ready to program in a whole new way!

Functional Programming in Java will help you quickly get on top of the new, essential Java 8 language features and the functional style that will change and improve your code. This short, targeted book will help you make the paradigm shift from the old imperative way to a less error-prone, more elegant, and concise coding style that's also a breeze to parallelize. You'll explore the syntax and semantics of lambda expressions, method and constructor references, and functional interfaces. You'll design and write applications better using the new standards in Java 8 and the JDK.

[Get it Now!](#)

One Response to "Introduction to PostgreSQL PL/java"



Sharon Hershon

January 15th, 2013 at 5:29 am

First, thank you for this article. There's nothing as comprehensive as this out there.

[Reply](#)

Leave a Reply

Name (Required)

Mail (will not be published) (Required)

Website

× 4 = eight

☐ Notify me of followup comments via e-mail

☒ Sign me up for the newsletter!

Submit Comment

Knowledge Base

[Examples](#)

[Resources](#)

[Tutorials](#)

[Whitepapers](#)

Partners

[Mkyong](#)

The Code Geeks Network

[Java Code Geeks](#)

[.NET Code Geeks](#)

[Web Code Geeks](#)

Hall Of Fame

[“Android Full Application Tutorial” series](#)

[GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial](#)

[Android Game Development Tutorials](#)

[Android Google Maps Tutorial](#)

[Android Location Based Services Application – GPS location](#)

[Funny Source Code Comments](#)

[Java Best Practices – Vector vs ArrayList vs HashSet](#)

[Android JSON Parsing with Gson Tutorial](#)

[Android Quick Preferences Tutorial](#)

About Java Code Geeks

JCGs (Java Code Geeks) is an independent online community creating the ultimate Java to Java developers resource including technical architect, technical team lead (senior developer), junior developers alike. JCGs serve the Java, SOA, Agile communities with daily news written by domain experts, announcements, code snippets and open source projects.

Java Code Geeks and all content copyright © 2010-2014, Exelixis Media Ltd | [Terms of Use](#)

All trademarks and registered trademarks appearing on Java Code Geeks are the property of their respective owners.

Java is a trademark or registered trademark of Oracle Corporation in the United States and other countries.

Java Code Geeks is not connected to Oracle Corporation and is not sponsored by Oracle Corporation.