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

```
# apt-get install postgresql-9.1
# apt-get install postgresql-9.1-pljava-gcj

$ createdb template_java
$ psql -d template_java -c 'update db_database set datistemplate='t' where datnam='template_java''
$ 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:

```
CREATE FUNCTION getsysprop(VARCHAR) RETURNS VARCHAR
  AS 'java.lang.System.getProperty'
  LANGUAGE java;

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

```
package sandbox;

```

```
public class PLJava {
    public static String hello(String name) {
        if (name == null) {
            return null;
        }

        return 'Hello, ' + name + '!';
    }
}
```

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

```
--
-- load java library
--
-- parameters:
--   url_path - where the library is located
--   url_name - how the library is referred to later
--   deploy   - should the deployment descriptor be used?
--
select sqlj.install_jar('file:///tmp/sandbox.jar', 'sandbox', true);

--
-- set classpath to include new library.
--
-- parameters
--   schema    - schema (or database) name
--   classpath - colon-separated list of url_names.
--
select sqlj.set_classpath('mydb', 'sandbox');

-- -------------------
-- other procedures --
-- -------------------

--
-- reload java library
--
select sqlj.replace_jar('file:///tmp/sandbox.jar', 'sandbox', true);

--
-- remove java library
--
-- parameters:
--   url_name - how the library is referred to later
--   undeploy - should the deployment descriptor be used?
--
select sqlj.remove_jar('sandbox', true);

--
-- list classpath
--
select sqlj.get_classpath('mydb');

--
```

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.

```
--
-- create function
--
CREATE FUNCTION mydb.hello(varchar) RETURNS varchar
  AS 'sandbox.PLJava.hello'
  LANGUAGE java;

--
-- drop this function
--
DROP FUNCTION mydb.hello(varchar);

--
```

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.

```
SQLActions[] = {
  'BEGIN INSTALL
     CREATE FUNCTION javatest.hello(varchar)
       RETURNS varchar
       AS 'sandbox.PLJava.hello'
       LANGUAGE java;
   END INSTALL',
  'BEGIN REMOVE
     DROP FUNCTION javatest.hello(varchar);
   END REMOVE'
}
```

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

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jar-plugin</artifactId>
   <version>2.3.1</version>
   <configuration>
      <archive>
         <manifestSections>
            <manifestSection>
               <name>postgresql.ddr</name> <!-- filename -->
               <manifestEntries>
                  <SQLJDeploymentDescriptor>TRUE</SQLJDeploymentDescriptor>
               </manifestEntries>
            </manifestSection>
         </manifestSections>
      </archive>
   </configuration>
</plugin>
```

The database will now know about our methods as they areinstalled 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.

```
public static Iterator<String> colors() {

    List<String> colors = Arrays.asList('red', 'green', 'blue');
    return colors.iterator();
}
```

and

```
CREATE FUNCTION javatest.colors()
    RETURNS SETOF varchar
    AS 'sandbox.PLJava.colors'
    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 implemention should either check this or use BigInteger.

```
public static Iterator seq(int start) {
    Iterator iter = null;
    try {
        iter = new SeqIterator(start);
    } catch (IllegalArgumentException e) {
        // should log error...
    }
    return iter;
}

public static class SeqIterator implements Iterator {
    private int next;
    private boolean done = false;

    public SeqIterator(int start) {
        if (start <= 0) {
            throw new IllegalArgumentException();
        }
        this.next = start;
    }

    @Override
    public boolean hasNext() {
        return !done;
    }

    @Override
```

```
    public Integer next() {
        int value = next;
        next = (next % 2 == 0) ? next / 2 : 3 * next + 1;
        done = (value == 1);
        return value;
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

```
CREATE FUNCTION javatest.seq(int)
    RETURNS SETOF int
    AS 'sandbox.PLJava.seq'
    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.

```
public static boolean singleWord(ResultSet receiver) throws SQLException {
    receiver.updateString('English', 'hello');
    receiver.updateString('Spanish', 'hola');
    return true;
}
```

and

```
CREATE TYPE word AS (
    English varchar,
    Spanish varchar);

CREATE FUNCTION javatest.single_word()
    RETURNS word
    AS 'sandbox.PLJava.singleWord'
    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.

```
public static ResultSetProvider listWords() {
     return new WordProvider();
 }

 public static class WordProvider implements ResultSetProvider {
     private final Map<String,String> words = new HashMap<String,String>();
     private final Iterator<String> keys;

     public WordProvider() {
         words.put('one', 'uno');
         words.put('two', 'dos');
         words.put('three', 'tres');
         words.put('four', 'quatro');
         keys = words.keySet().iterator();
     }

     @Override
     public boolean assignRowValues(ResultSet receiver, int currentRow)
             throws SQLException {

         if (!keys.hasNext()) {
             return false;
         }
         String key = keys.next();
         receiver.updateString('English', key);
         receiver.updateString('Spanish', words.get(key));
         return true;
     }

     @Override
     public void close() throws SQLException {
     }
 }
```

and

```
CREATE FUNCTION javatest.list_words()
  RETURNS SETOF word
  AS 'sandbox.PLJava.listWords'
  IMMUTABLE LANGUAGE java;
```

*org.postgresql.pljava.ResultSetHandle*

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

```
public static ResultSetHandle listUsers() {
    return new UsersHandle();
}

public static class UsersHandle implements ResultSetHandle {
    private Statement stmt;

    @Override
    public ResultSet getResultSet() throws SQLException {
        stmt = DriverManager.getConnection('jdbc:default:connection').createStatement();
        return stmt.executeQuery('SELECT * FROM pg_user');
    }

    @Override
    public void close() throws SQLException {
        stmt.close();
    }
}
```

and

```
CREATE FUNCTION javatest.list_users()
    RETURNS SETOF pg_user
    AS 'sandbox.PLJava.listUsers'
    LANGUAGE java;
```

**The Interfaces**

I have been unable 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*

```
// Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
 // Distributed under the terms shown in the file COPYRIGHT
 // found in the root folder of this project or at
 // http://eng.tada.se/osprojects/COPYRIGHT.html

package org.postgresql.pljava;

import java.sql.ResultSet;
import java.sql.SQLException;


 // An implementation of this interface is returned from functions and procedures
 // that are declared to return <code>SET OF</code> a complex type.    //Functions that
 // return <code>SET OF</code> a simple type should simply return an
 // {@link java.util.Iterator Iterator}.
 // @author Thomas Hallgren

public interface ResultSetProvider
{

  // This method is called once for each row that should be returned from
  // a procedure that returns a set of rows. The receiver
  // is a {@link org.postgresql.pljava.jdbc.SingleRowWriter SingleRowWriter}
  // writer instance that is used for capturing the data for the row.
  // @param receiver Receiver of values for the given row.
  // @param currentRow Row number. First call will have row number 0.
  // @return <code>true</code> if a new row was provided,   <code>false</code>
  // if not (end of data).
  // @throws SQLException

 boolean assignRowValues(ResultSet receiver, int currentRow)
 throws SQLException;


  // Called after the last row has returned or when the query evaluator dec       ides
  // that it does not need any more rows.
  //
 void close()
 throws SQLException;
}
```

*ResultSetHandle*

```
// Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
// Distributed under the terms shown in the file COPYRIGHT
// found in the root directory of this distribution or at
// http://eng.tada.se/osprojects/COPYRIGHT.html

package org.postgresql.pljava;

import java.sql.ResultSet;
import java.sql.SQLException;


 // An implementation of this interface is returned from functions and procedures
 // that are declared to return <code>SET OF</code> a complex type in the form
 // of a {@link java.sql.ResultSet}. The primary motivation for this interface is
 // that an implementation that returns a ResultSet must be able to close the
 // connection and statement when no more rows are requested.
```

```
 // @author Thomas Hallgren

public interface ResultSetHandle
{

  // An implementation of this method will probably execute a query
  // and return the result of that query.
  // @return The ResultSet that represents the rows to be returned.
  // @throws SQLException

 ResultSet getResultSet()
 throws SQLException;


 // Called after the last row has returned or when the query evaluator decides
 // that it does not need any more rows.

 void close()
 throws SQLException;
}
```

**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 a 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.

```
public class AuditTrigger {

    public static void auditFoobar(TriggerData td) throws SQLException {

        Connection conn = DriverManager
                .getConnection('jdbc:default:connection');
        PreparedStatement ps = conn
                .prepareStatement('insert into javatest.foobar_audit(what, whenn, data) values (?,
?, ?::xml)');

        if (td.isFiredByInsert()) {
            ps.setString(1, 'INSERT');
        } else if (td.isFiredByUpdate()) {
            ps.setString(1, 'UPDATE');
        } else if (td.isFiredByDelete()) {
            ps.setString(1, 'DELETE');
        }
        ps.setTimestamp(2, new Timestamp(System.currentTimeMillis()));

        ResultSet rs = td.getNew();
        if (rs != null) {
            ps.setString(3, toXml(rs));
        } else {
            ps.setNull(3, Types.VARCHAR);
```

```
        }

        ps.execute();
        ps.close();
    }

    // simple marshaler. We could use jaxb or similar library
    static String toXml(ResultSet rs) throws SQLException {
        String foo = rs.getString(1);
        if (rs.wasNull()) {
            foo = '';
        }
        String bar = rs.getString(2);
        if (rs.wasNull()) {
            bar = '';
        }
        return String.format('<my-class><foo>%s</foo><bar>%s</bar></my-class>', foo, bar);
    }
}
```

```
CREATE TABLE javatest.foobar (
     foo   varchar(10),
     bar   varchar(10)
);

CREATE TABLE javatest.foobar_audit (
     what  varchar(10) not null,
     whenn timestamp not null,
     data  xml
);

CREATE FUNCTION javatest.audit_foobar()
    RETURNS trigger
    AS 'sandbox.AuditTrigger.auditFoobar'
    LANGUAGE 'java';

CREATE TRIGGER foobar_audit
    AFTER INSERT OR UPDATE OR DELETE ON javatest.foobar
    FOR EACH ROW
    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*

```
// Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
 // Distributed under the terms shown in the file COPYRIGHT
 // found in the root folder of this project or at
 // http://eng.tada.se/osprojects/COPYRIGHT.html

package org.postgresql.pljava;

import java.sql.ResultSet;
import java.sql.SQLException;


 // The SQL 2003 spec. does not stipulate a standard way of mapping
 // triggers to functions. The PLJava mapping use this interface. All
 // functions that are intended to be triggers must be public, static,
 // return void, and take a <code>TriggerData</code> as their argument.
 //
 // @author Thomas Hallgren

public interface TriggerData
{

  // Returns the ResultSet that represents the new row. This ResultSet wil
  // be null for delete triggers and for triggers that was fired for
  // statement.
        //The returned set will be updateable and positioned on a
  // valid row. When the trigger call returns, the trigger manager will se
  // the changes that has been made to this row and construct a new tuple
  // which will become the new or updated row.
  //
  // @return An updateable <code>ResultSet</code> containing one row or
  // null
  // @throws SQLException
  //             if the contained native buffer has gone stale.
  //
 ResultSet getNew() throws SQLException;


  // Returns the ResultSet that represents the old row. This ResultSet wil
  // be null for insert triggers and for triggers that was fired for
  // statement.The returned set will be read-only and positioned on a
  // valid row.
  //
```

```
  // @return A read-only ResultSet containing one row or
  //         null.
  // @throws SQLException
  //             if the contained native buffer has gone stale.
  //
 ResultSet getOld() throws SQLException;

 //
 // Returns the arguments for this trigger (as declared in the <code>CREAT        // E
TRIGGER</code>
 // statement. If the trigger has no arguments, this method will return an
 // array with size 0.
 //
 // @throws SQLException
 //             if the contained native buffer has gone stale.

 String[] getArguments() throws SQLException;
```

```
// Returns the name of the trigger (as declared in theCREATE TRIGGER
  // statement).
  //
 // @throws SQLException
  //             if the contained native buffer has gone stale.
  //
 String getName() throws SQLException;
/**
//Returns the name of the table for which this trigger was created (as
//* declared in the <code>CREATE TRIGGER</code statement). * * @throws SQLException* if the
contained native buffer has gone stale.
String getTableName() throws SQLException;
/// Returns the name of the schema of the table for which this trigger was created (as * declared
in the <code>CREATE TRIGGER</code statement).
//@throws SQLException * if the contained native buffer has gone stale. */

String getSchemaName() throws SQLException;
// Returns <code>true</code> if the trigger was fired after the statement  or row action that it is
associated with.
//@throws SQLException * if the contained native buffer has gone stale.

boolean isFiredAfter() throws SQLException;
//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. */
boolean isFiredBefore() throws SQLException;
//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.
*/
boolean isFiredForEachRow() throws SQLException;
//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. */
boolean isFiredForStatement() throws SQLException;
//Returns <code>true</code> if this trigger was fired by a <code>DELETE</code>. * * @throws
SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByDelete() throws SQLException;
//Returns <code>true</code> if this trigger was fired by an //<code>INSERT</code>. * * @throws
SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByInsert() throws SQLException;
//Returns <code>true</code> if this trigger was fired by an //<code>UPDATE</code>. * * @throws
SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByUpdate() throws SQLException;

// Returns the name of the table for which this trigger was created (as
// declared in the <code>CREATE TRIGGER</code statement). * * @throws //SQLException* if the
contained native buffer has gone stale. */
String getTableName() throws SQLException;
// Returns the name of the schema of the table for which this trigger was created (as / declared in
the <code>CREATE TRIGGER</code statement). * * @throws //SQLException * if the contained native
buffer has gone stale. */
String getSchemaName() throws SQLException;
//Returns <code>true</code> if the trigger was fired after the statement // or row action that it
is associated with. * * @throws SQLException * if the //contained native buffer has gone stale. */
boolean isFiredAfter() throws SQLException;
// 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.
*/
boolean isFiredBefore() throws SQLException;
// 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.
*/
boolean isFiredForEachRow() throws SQLException;
// 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.
*/
boolean isFiredForStatement() throws SQLException;
// Returns <code>true</code> if this trigger was fired by a //<code>DELETE</code>. * * @throws
SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByDelete() throws SQLException;
// Returns <code>true</code> if this trigger was fired by an //<code>INSERT</code>. * * @throws
SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByInsert() throws SQLException;
// Returns <code>true</code> if this trigger was fired by an //<code>UPDATE</code>. * * @throws
SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByUpdate() throws SQLException; }/**
// Returns the name of the table for which this trigger was created (as
// declared in the <code>CREATE TRIGGER</code statement). * * @throws //SQLException* if the
contained native buffer has gone stale. */
String getTableName() throws SQLException;
// Returns the name of the schema of the table for which this trigger was created (as // declared
in the <code>CREATE TRIGGER</code statement). * * @throws //SQLException * if the contained native
buffer has gone stale. */
String getSchemaName() throws SQLException;
/// Returns <code>true</code> if the trigger was fired after the //statement * or row action that
it is associated with. * * @throws SQLException * if //the contained native buffer has gone stale.
*/
boolean isFiredAfter() throws SQLException;
```

```
// 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. */
boolean isFiredBefore() throws SQLException;
// 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. */
boolean isFiredForEachRow() throws SQLException;
// 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. */
boolean isFiredForStatement() throws SQLException;
// Returns <code>true</code> if this trigger was fired by a //<code>DELETE</code>. * * @throws SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByDelete() throws SQLException;
// Returns <code>true</code> if this trigger was fired by an //<code>INSERT</code>. * * @throws SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByInsert() throws SQLException;
// Returns <code>true</code> if this trigger was fired by an //<code>UPDATE</code>. * * @throws SQLException * if the contained native //buffer has gone stale. */
boolean isFiredByUpdate() throws SQLException; }
```

*TriggerException*

```
// Copyright (c) 2004, 2005, 2006 TADA AB - Taby Sweden
 // Distributed under the terms shown in the file COPYRIGHT
 // found in the root folder of this project or at
 // http://eng.tada.se/osprojects/COPYRIGHT.html

package org.postgresql.pljava;

import java.sql.SQLException;


 // An exception specially suited to be thrown from within a method
 // designated to be a trigger function. The message generated by
 // this exception will contain information on what trigger and
 // what relation it was that caused the exception
 //
 // @author Thomas Hallgren

public class TriggerException extends SQLException
{
    private static final long serialVersionUID = 5543711707414329116L;

    private static boolean s_recursionLock = false;

    public static final String TRIGGER_ACTION_EXCEPTION = '09000';

    private static final String makeMessage(TriggerData td, String message)
    {
        StringBuffer bld = new StringBuffer();
        bld.append('In Trigger ');
        if(!s_recursionLock)
        {
            s_recursionLock = true;
            try
            {
                bld.append(td.getName());
                bld.append(' on relation ');
                bld.append(td.getTableName());
            }
            catch(SQLException e)
            {
                bld.append('(exception while generating exception message)');
            }
            finally
            {
                s_recursionLock = false;
            }
        }
        if(message != null)
        {
            bld.append(': ');
            bld.append(message);
        }
        return bld.toString();
    }


     // Create an exception based on the <code>TriggerData</code> that was
     // passed to the trigger method.
     // @param td The <code>TriggerData</code> that was passed to the trigger
     // method.

    public TriggerException(TriggerData td)
    {
        super(makeMessage(td, null), TRIGGER_ACTION_EXCEPTION);
    }


     // Create an exception based on the <code>TriggerData</code> that was
     // passed to the trigger method and an additional message.
     // @param td The <code>TriggerData</code> that was passed to the trigger
     // method.
     // @param reason An additional message with info about the exception.

    public TriggerException(TriggerData td, String reason)
    {
        super(makeMessage(td, reason), TRIGGER_ACTION_EXCEPTION);
    }
}
```

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 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 do to 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('demominator 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;
```

```
    }

    public Rational(int numerator, int denominator, String typeName)
            throws SQLException {
        this(numerator, denominator);
        this.typeName = typeName;
    }

    public String getSQLTypeName() {
        return typeName;
    }

    public void readSQL(SQLInput stream, String typeName) throws SQLException {
        this.numerator = stream.readLong();
        this.denominator = stream.readLong();
        this.typeName = typeName;
    }

    public void writeSQL(SQLOutput stream) throws SQLException {
        stream.writeLong(numerator);
        stream.writeLong(denominator);
    }

    public String toString() {
        String value = null;
        if (denominator == 1) {
            value = String.valueOf(numerator);
        } else {
            value = String.format('%d/%d', numerator, denominator);
        }
        return value;
    }

    /*
     * Meaningful code that actually does something with this type was
     * intentionally left out.
     */
}
```

and

```
/* The shell type */
CREATE TYPE javatest.rational;

/* The scalar input function */
CREATE FUNCTION javatest.rational_in(cstring)
  RETURNS javatest.rational
  AS 'UDT[sandbox.Rational] input'
  LANGUAGE java IMMUTABLE STRICT;

/* The scalar output function */
CREATE FUNCTION javatest.rational_out(javatest.rational)
  RETURNS cstring
  AS 'UDT[sandbox.Rational] output'
  LANGUAGE java IMMUTABLE STRICT;

/* The scalar receive function */
CREATE FUNCTION javatest.rational_recv(internal)
  RETURNS javatest.rational
  AS 'UDT[sandbox.Rational] receive'
  LANGUAGE java IMMUTABLE STRICT;

/* The scalar send function */
CREATE FUNCTION javatest.rational_send(javatest.rational)
  RETURNS bytea
  AS 'UDT[sandbox.Rational] send'
  LANGUAGE java IMMUTABLE STRICT;

CREATE TYPE javatest.rational (
  internallength = 16,
  input = javatest.rational_in,
  output = javatest.rational_out,
  receive = javatest.rational_recv,
  send = javatest.rational_send,
  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.

```
public static Rational newInstance(String input) throws SQLException {
    if (input == null) {
        return null;
    }
    return parse(input, 'javatest.rational');
}

public static Rational newInstance(int value) throws SQLException {
    return new Rational(value);
}

public static Rational newInstance(Integer value) throws SQLException {
    if (value == null) {
        return null;
    }
    return new Rational(value.intValue());
}

public static Rational newInstance(long value) throws SQLException {
    return new Rational(value);
}

public static Rational newInstance(Long value) throws SQLException {
    if (value == null) {
        return null;
    }
    return new Rational(value.longValue());
}


public static Double value(Rational value) throws SQLException {
    if (value == null) {
        return null;
    }
    return value.doubleValue();
}
```

and

```
CREATE FUNCTION javatest.rational_string_as_rational(varchar) RETURNS javatest.rational
      AS 'sandbox.Rational.newInstance'
      LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_int_as_rational(int4) RETURNS javatest.rational
      AS 'sandbox.Rational.newInstance'
      LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_long_as_rational(int8) RETURNS javatest.rational
      AS 'sandbox.Rational.newInstance'
      LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_as_double(javatest.rational) RETURNS float8
      AS 'sandbox.Rational.value'
      LANGUAGE JAVA IMMUTABLE STRICT;

CREATE CAST (varchar AS javatest.rational)
    WITH FUNCTION javatest.rational_string_as_rational(varchar)
    AS ASSIGNMENT;

CREATE CAST (int4 AS javatest.rational)
    WITH FUNCTION javatest.rational_int_as_rational(int4)
    AS ASSIGNMENT;

CREATE CAST (int8 AS javatest.rational)
    WITH FUNCTION javatest.rational_long_as_rational(int8)
    AS ASSIGNMENT;

CREATE CAST (javatest.rational AS float8)
    WITH FUNCTION javatest.rational_as_double(javatest.rational)
    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.

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

```
    } else if (q == null) {
        return -1;
    }
    BigInteger l =
BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator()));
    BigInteger r =
BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator()));
    return l.compareTo(r);
}

public static Rational min(Rational p, Rational q) {
    if ((p == null) || (q == null)) {
        return null;
    }
    return (p.compareTo(q) <= 0) ? p : q;
}

public static Rational max(Rational p, Rational q) {
    if ((p == null) || (q == null)) {
        return null;
    }
    return (q.compareTo(p) < 0) ? p : q;
}

public static Rational add(Rational p, Rational q) throws SQLException {
    if ((p == null) || (q == null)) {
        return null;
    }
    BigInteger n =
BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator())).add(
            BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator())));
    BigInteger d =
BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
    BigInteger gcd = n.gcd(d);
    n = n.divide(gcd);
    d = d.divide(gcd);
    return new Rational(n.longValue(), d.longValue());
}
```

and

```
CREATE FUNCTION javatest.min(javatest.rational, javatest.rational) RETURNS javatest.rational
    AS 'sandbox.Rational.min'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.max(javatest.rational, javatest.rational) RETURNS javatest.rational
    AS 'sandbox.Rational.max'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE AGGREGATE min(javatest.rational) (
  sfunc = javatest.min,
  stype = javatest.rational
);

CREATE AGGREGATE max(javatest.rational) (
  sfunc = javatest.max,
  stype = javatest.rational
);

CREATE AGGREGATE sum(javatest.rational) (
  sfunc = javatest.add,
  stype = javatest.rational
);
```

**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.

```
public class Rational implements UserType, Serializable {
    private final int[] sqlTypesSupported = new int[] { Types.OTHER };
    private long numerator;
    private long denominator;

    public Rational() {
        numerator = 0;
        denominator = 1;
    }

    public Rational(long numerator, long denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public long getNumerator() {
        return numerator;
    }

    public long getDenominator() {
        return denominator;
    }

```

```
    @Override
    public Object assemble(Serializable cached, Object owner)
            throws HibernateException {
        if (!(cached instanceof Rational)) {
            throw new HibernateException('invalid argument');
        }
        Rational r = (Rational) cached;
        return new Rational(r.getNumerator(), r.getDenominator());
    }

    @Override
    public Serializable disassemble(Object value) throws HibernateException {
        if (!(value instanceof Rational)) {
            throw new HibernateException('invalid argument');
        }
        return (Rational) value;
    }

    @Override
    public Object deepCopy(Object value) throws HibernateException {
        if (value == null) {
            return null
        }
        if (!(value instanceof Rational)) {
            throw new HibernateException('invalid argument');
        }
        Rational v = (Rational) value;
        return new Rational(v.getNumerator(), v.getDenominator());
    }

    @Override
    public boolean isMutable() {
        return true;
    }

    //
    // important: PGobject is postgresql-specific
    //
    @Override
    public Object nullSafeGet(ResultSet rs, String[] names, Object owners)
            throws HibernateException, SQLException {
        PGobject pgo = (PGobject) rs.getObject(names[0]);
        if (rs.wasNull()) {
            return null;
        }
        TheDbRationalType r = TheDbRationalType.parse(pgo.getValue(), 'rational');
        return new Rational(r.getNumerator(), r.getDenominator());
    }

    //
    // important: using Types.OTHER may be postgresql-specific
    //
    @Override
    public void nullSafeSet(PreparedStatement ps, Object value, int index)
            throws HibernateException, SQLException {
        if (value == null) {
            ps.setNull(index, Types.OTHER);
        } else if (!(value instanceof Rational)) {
            throw new HibernateException('invalid argument');
        } else {
            Rational t = (Rational) value;
            ps.setObject(index,
                    new TheDbRationalType(t.getNumerator(), t.getDenominator()), Types.OTHER);
        }
    }

    @Override
    public Object replace(Object original, Object target, Object owner)
            throws HibernateException {
        if (!(original instanceof Rational)
                || !(target instanceof Rational)) {
            throw new HibernateException('invalid argument');
        }
        Rational r = (Rational) original;
        return new Rational(r.getNumerator(), r.getDenominator());
    }

    @Override
    public Class returnedClass() {
        return Rational.class;
    }

    @Override
    public int[] sqlTypes() {
        return sqlTypesSupported;
    }

    @Override
    public String toString() {
        String value = '';
        if (denominator == 1) {
            value = String.valueOf(numerator);
        } else {
            value = String.format('%d/%d', numerator, denominator);
        }
        return value;
    }

    // for UserType
    @Override
    public int hashCode(Object value) {
        Rational r = (Rational) value;
        return (int) (31 * r.getNumerator() + r.getDenominator());
```

```
    }

    @Override
    public int hashCode() {
        return hashCode(this);
    }

    // for UserType
    @Override
    public boolean equals(Object left, Object right) {
        if (left == right) {
            return true;
        }
        if ((left == null) || (right == null)) {
            return false;
        }
        if (!(left instanceof Rational) || !(right instanceof Rational)) {
            return false;
        }

        Rational l = (Rational) left;
        Rational r = (Rational) right;
        return (l.getNumerator() == r.getNumerator())
                && (l.getDenominator() == r.getDenominator());
    }

    @Override
    public boolean equals(Object value) {
        return equals(this, value);
    }
}
```

*CustomTypes.hbm.xml*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
        '-//Hibernate/Hibernate Mapping DTD 3.0//EN'
        'http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd '>

<hibernate-mapping>

    <typedef name='javatest.rational' class='sandbox.RationalType'/>

</hibernate-mapping>
```

*TestTable.hbm.xml*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
        '-//Hibernate/Hibernate Mapping DTD 3.0//EN'
        'http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd '>

<hibernate-mapping>

    <class name='sandbox.TestTable' table='test_table'>
        <id name='id'/>
        <property name='value' type='javatest.rational' />
    </class>

</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

```
public static Rational negate(Rational p) throws SQLException {
    if (p == null) {
        return null;
    }
    return new Rational(-p.getNumerator(), p.getDenominator());
}

public static Rational add(Rational p, Rational q) throws SQLException {
    if ((p == null) || (q == null)) {
        return null;
    }
    BigInteger n =
BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator())).add(
            BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator())));
    BigInteger d =
BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
    BigInteger gcd = n.gcd(d);
    n = n.divide(gcd);
    d = d.divide(gcd);
    return new Rational(n.longValue(), d.longValue());
}

public static Rational subtract(Rational p, Rational q) throws SQLException {
    if ((p == null) || (q == null)) {
        return null;
    }
    BigInteger n =
BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator())).subtract(
            BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator())));
    BigInteger d =
BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
    BigInteger gcd = n.gcd(d);
    n = n.divide(gcd);
```

```
    d = d.divide(gcd);
    return new Rational(n.longValue(), d.longValue());
}

public static Rational multiply(Rational p, Rational q) throws SQLException {
    if ((p == null) || (q == null)) {
        return null;

    }
    BigInteger n =
BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getNumerator()));
    BigInteger d =
BigInteger.valueOf(p.getDenominator()).multiply(BigInteger.valueOf(q.getDenominator()));
    BigInteger gcd = n.gcd(d);
    n = n.divide(gcd);
    d = d.divide(gcd);
    return new Rational(n.longValue(), d.longValue());
}
```

and

```
CREATE FUNCTION javatest.rational_negate(javatest.rational) RETURNS javatest.rational
    AS 'sandbox.Rational.negate'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_add(javatest.rational, javatest.rational)
    RETURNS javatest.rational
    AS 'sandbox.Rational.add'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_subtract(javatest.rational, javatest.rational)
    RETURNS javatest.rational
    AS 'sandbox.Rational.subtract'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_multiply(javatest.rational, javatest.rational)
    RETURNS javatest.rational
    AS 'sandbox.Rational.multiply'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_divide(javatest.rational, javatest.rational)
    RETURNS javatest.rational
    AS 'sandbox.Rational.divide'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE OPERATOR - (
   rightarg = javatest.rational, procedure.rational_negate
);

CREATE OPERATOR + (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_add,
   commutator = +
);

CREATE OPERATOR - (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure =
javatest.rational_subtract
);

CREATE OPERATOR * (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_divide,
   commutator = *
);

CREATE OPERATOR / (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_divide
);
```

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.

```
public static int compare(Rational p, Rational q) {
    if (p == null) {
        return 1;
    } else if (q == null) {
        return -1;
    }
    BigInteger l =
BigInteger.valueOf(p.getNumerator()).multiply(BigInteger.valueOf(q.getDenominator()));
    BigInteger r =
```

```
    BigInteger.valueOf(q.getNumerator()).multiply(BigInteger.valueOf(p.getDenominator()));
    return l.compareTo(r);
}

public int compareTo(Rational p) {
    return compare(this, p);
}

public static int compare(Rational p, double q) {
    if (p == null) {
        return 1;
    }
    double d = p.doubleValue();
    return (d < q) ? -1 : ((d == q) ? 0 : 1);
}

public int compareTo(double q) {
    return compare(this, q);
}

public static boolean lessThan(Rational p, Rational q) {
    return compare(p, q) < 0;
}

public static boolean lessThanOrEquals(Rational p, Rational q) {
    return compare(p, q) <= 0;
}

public static boolean equals(Rational p, Rational q) {
    return compare(p, q) = 0;
}

public static boolean greaterThan(Rational p, Rational q) {
    return compare(p, q) > 0;
}


public static boolean lessThan(Rational p, double q) {
    if (p == null) {
        return false;
    }
    return p.compareTo(q) < 0;
}

public static boolean lessThanOrEquals(Rational p, double q) {
    if (p == null) {
        return false;
    }
    return p.compareTo(q) = 0;
}

public static boolean greaterThan(Rational p, double q) {
    if (p == null) {
        return true;
    }
    return p.compareTo(q) > 0;
}
```

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

```
CREATE FUNCTION javatest.rational_lt(javatest.rational, javatest.rational)
    RETURNS bool
    AS 'sandbox.Rational.lessThan'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_le(javatest.rational, javatest.rational)
    RETURNS bool
    AS 'sandbox.Rational.lessThanOrEquals'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_eq(javatest.rational, javatest.rational)
    RETURNS bool
    AS 'sandbox.Rational.equals'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_ge(javatest.rational, javatest.rational)
    RETURNS bool
    AS 'sandbox.Rational.greaterThanOrEquals'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_gt(javatest.rational, javatest.rational)
    RETURNS bool
    AS 'sandbox.Rational.greaterThan'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_cmp(javatest.rational, javatest.rational)
    RETURNS int
    AS 'sandbox.Rational.compare'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_lt(javatest.rational, float8)
    RETURNS bool
    AS 'sandbox.Rational.lessThan'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_le(javatest.rational, float8)
    RETURNS bool
    AS 'sandbox.Rational.lessThanOrEquals'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_eq(javatest.rational, float8)
    RETURNS bool
    AS 'sandbox.Rational.equals'
```

```
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_ge(javatest.rational, float8)
    RETURNS bool
    AS 'sandbox.Rational.greaterThanOrEquals'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE FUNCTION javatest.rational_gt(javatest.rational, float8)
    RETURNS bool
    AS 'sandbox.Rational.greaterThan'
    LANGUAGE JAVA IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel, merges
);

CREATE OPERATOR <= (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_le,
   commutator = >= , negator = > ,
   restrict = scalarltsel, join = scalarltjoinsel, merges
);

CREATE OPERATOR = (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_eq,
   commutator = = , negator = <>, hashes, merges
);

CREATE OPERATOR >= (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_lt,
   commutator = <= , negator = < ,
   restrict = scalarltsel, join = scalarltjoinsel, merges
);

CREATE OPERATOR > (
   leftarg = javatest.rational, rightarg = javatest.rational, procedure = javatest.rational_le,
   commutator = <= , negator = < ,
   restrict = scalargtsel, join = scalargtjoinsel, merges
);

CREATE OPERATOR < (
   leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_lt,
   commutator = > , negator = >=
);

CREATE OPERATOR <= (
   leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_le,
   commutator = >= , negator = >
);

CREATE OPERATOR = (
   leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_eq,
   commutator = = , negator = <>
);

CREATE OPERATOR >= (
   leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_ge,
   commutator = <= , negator = <
);

CREATE OPERATOR > (
   leftarg = javatest.rational, rightarg = float8, procedure = javatest.rational_gt,
   commutator = < , negator = <=
);
```

*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.

```
  -- btree join
CREATE OPERATOR CLASS rational_ops
    DEFAULT FOR TYPE javatest.rational USING btree AS
      OPERATOR        1       < ,
      OPERATOR        2       <= ,
      OPERATOR        3       = ,
      OPERATOR        4       >= ,
      OPERATOR        5       > ,
      FUNCTION        1       javatest.rational_cmp(javatest.rational, javatest.rational);

  -- hash join
 CREATE OPERATOR CLASS rational_ops
    DEFAULT FOR TYPE javatest.rational USING hash AS
      OPERATOR        1       = ,
      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 . java
- CREATE OPERATOR (PostgreSQL)
- CREATE OPERATOR CLASS (PostgreSQL)
- CREATE OPERATOR FAMILY (PostgreSQL)
- Operator Optimization (PostgreSQL)
- Interfacing Extensions To Indexes (PostreSQL)
- 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



## 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 com
creating the ultimate Java to Java developers resource
technical architect, technical team lead (senior develope
junior developers alike. JCGs serve the Java, SOA, Agil
communities with daily news written by domain experts,
announcements, code snippets and open source projec