Contents

Functions

A Java function is declared with the name of a class and a public static method on that class. The class will be resolved using the classpath that has been defined for the schema where the function is declared. If no classpath has been defined for that schema, the public schema is used. Please note that the *system classloader* will take precedence always. There is no way to override classes loaded with that loader.

The following function can be declared to access the static method getProperty on the java.lang.System class:

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

SELECT getsysprop('java.version');
```

Both the parameters and the return value can be explicitly stated so the above example could also have been written:

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

This way of declaring the function is useful when the default mapping is inadequate. PL/Java will use a standard PostgreSQL explicit cast when the SQL type of the parameter or return value does not correspond to the Java type defined in the mapping.

Note: the "explicit cast" here referred to is not accomplished by creating an actual SQL CAST expression, but by (mostly) equivalent means. At the time of this writing, two special cases are not yet implemented.

SQL generation

The simplest way to write the SQL function declaration that corresponds to your Java code is to have the Java compiler do it for you:

```
public class Hello {
  @Function
  public static String hello(String toWhom) {
    return "Hello, " + toWhom + "!";
  }
}
```

When this function is compiled, a "deployment descriptor" containing the right SQL function declaration is also produced. When it is included in a jar file with the compiled code, PL/Java's sqlj.install_jar function will create the SQL function declaration at the same time it loads the jar. See the full hello world example for more.

Set-returning functions

Returning sets is tricky. You don't want to first build a set and then return it, since large sets would eat excessive resources. It's better to produce one row at a time. Incidentally, that's exactly what the PostgreSQL backend expects a function that RETURNS SETOF <type> to do. The <type> can be a *scalar type* such as an *int*, *float* or *varchar*, it can be a *complex type*, or a *RECORD*.

Returning a SETOF <scalar type>

In order to return a set of a scalar type, you need create a Java method that returns an implementation the java.util.Iterator interface.

```
CREATE FUNCTION javatest.getNames()
  RETURNS SETOF varchar
  AS 'foo.fee.Bar.getNames'
  IMMUTABLE LANGUAGE java;
```

The corresponding Java class:

```
package foo.fee;
import java.util.Iterator;

import org.postgresql.pljava.annotation.Function;
import static org.postgresql.pljava.annotation.Function.Effects.IMMUTABLE;

public class Bar
{
    @Function(schema="javatest", effects=IMMUTABLE)
    public static Iterator<String> getNames()
    {
        ArrayList<String> names = new ArrayList<>();
        names.add("Lisa");
        names.add("Bob");
        names.add("Bill");
        names.add("Sally");
        return names.iterator();
    }
}
```

Returning a SETOF <complex type>

A method returning a SETOF <complex type> must use either the interface org.postgresql.pljava.ResultSetProvider or org.postgresql.pljava.ResultSetHandle. The reason for having two interfaces is that they cater for optimal handling of two distinct use cases. The former is great when you want to dynamically create each row that is to be returned from the SETOF function. The latter makes sense when you want to return the result of an executed query.

Using the ResultSetProvider interface

This interface has two methods. The boolean assignRowValues(java.sql.ResultSet tupleBuilder, int rowNumbe and the void close() method. The PostgreSQL query evaluator will call the assignRowValues() repeatedly until it returns false or until the evaluator decides that it does not need any more rows. It will then call close().

You can use this interface the following way:

```
CREATE FUNCTION javatest.listComplexTests(int, int)
  RETURNS SETOF complexTest
  AS 'foo.fee.Fum.listComplexTest'
  IMMUTABLE LANGUAGE java;
```

The function maps to a static java method that returns an instance that implements the ResultSetProvider interface.

```
public class Fum implements ResultSetProvider
{
  private final int m_base;
  private final int m_increment;
  public Fum(int base, int increment)
  {
    m_base = base;
    m_increment = increment;
  }
  public boolean assignRowValues(ResultSet receiver, int currentRow)
  throws SQLException
  {
    // Stop when we reach 12 rows.
    //
    if(currentRow >= 12)
      return false;
    receiver.updateInt(1, m_base);
    receiver.updateInt(2, m_base + m_increment * currentRow);
    receiver.updateTimestamp(3, new Timestamp(System.currentTimeMillis()));
    return true;
  }
  public void close()
  {
    // Nothing needed in this example
  }
  @Function(effects=IMMUTABLE, schema="javatest", type="complexTest")
  public static ResultSetProvider listComplexTests(int base, int increment)
  throws SQLException
  {
    return new Fum(base, increment);
  }
```

```
}
```

The listComplexTests(int base, int increment) method is called once. It may return null if no results are available, or an instance of the ResultSetProvider. Here the Fum class implements this interface so it returns an instance of itself. The method assignRowValues(ResultSet receiver, int currentRow) will then be called repeatedly until it returns false. At that time, close() will be called.

The currentRow parameter can be a convenience in some cases, and unnecessary in others. It will be passed as zero on the first call, and incremented by one on each subsequent call. If the ResultSetProvider is returning results from some source (like an Iterator) that remembers its own position, it can simply ignore currentRow.

Using the ResultSetHandle interface

This interface is similar to the ResultSetProvider interface in that it has a close() method that will be called at the end. But instead of having the evaluator call a method that builds one row at a time, this method has a method that returns a ResultSet. The query evaluator will iterate over this set and deliver its contents, one tuple at a time, to the caller until a call to next() returns false or the evaluator decides that no more rows are needed.

Here is an example that executes a query using a statement that it obtained using the default connection. The SQL looks like this:

```
CREATE FUNCTION javatest.listSupers()
  RETURNS SETOF pg_user
  AS 'org.postgresql.pljava.example.Users.listSupers'
  LANGUAGE java;

CREATE FUNCTION javatest.listNonSupers()
  RETURNS SETOF pg_user
  AS 'org.postgresql.pljava.example.Users.listNonSupers'
  LANGUAGE java;
```

And here is the Java code:

```
public class Users implements ResultSetHandle
{
  private final String m_filter;
  private Statement m_statement;

  public Users(String filter)
  {
    m_filter = filter;
  }
```

```
  public ResultSet getResultSet()
  throws SQLException
  {
    m_statement = DriverManager.getConnection("jdbc:default:connection")
      .createStatement();
    return m_statement.executeQuery("SELECT * FROM pg_user WHERE " + m_filter);
  }

  public void close()
  throws SQLException
  {
    m_statement.close();
  }

  @Function(schema="javatest", type="pg_user")
  public static ResultSetHandle listSupers()
  {
    return new Users("usesuper = true");
  }

  @Function(schema="javatest", type="pg_user")
  public static ResultSetHandle listNonSupers()
  {
    return new Users("usesuper = false");
  }
}
```

Installing PL/Java

For the most current information on installing PL/Java, see the installation guide on the project information site.

Logging in PL/Java

PL/Java uses the standard java.util.logging.Logger Hence, you can write things like:

```
Logger.getAnonymousLogger().info(
    "Time is " + new Date(System.currentTimeMillis()));
```

At present, the logger is hardwired to a handler that maps the state of the PostgreSQL configuration setting log_min_messages to a valid Logger level and that outputs all messages using the backend function ereport().

Importantly, Java's Logger methods can quickly discard any message logged at

a finer level than the one that was mapped from PostgreSQL's setting *at the time PL/Java was first used in the current session*. Such messages never even get as far as ereport(), even if the PostgreSQL setting is changed later.

So, if expected messages from Java code are not showing up, be sure that the setting in PostgreSQL, at the time of PL/Java's first use in the session, is fine enough that Java will not throw the messages away. Once PL/Java has started, the settings can be changed as desired and will control, in the usual way, what ereport does with the messages PL/Java delivers to it.

Through PL/Java 1.5.0, only the log_min_messages setting is used to set that Java cutoff level. Starting with 1.5.1, the cutoff level in Java is set (still only once at PL/Java startup) based on the finer of log_min_messages and client_min_messages.

The following mapping applies between the Logger levels and the PostgreSQL backend levels:

java.util.logging.Level

PostgreSQL level

SEVERE

ERROR

WARNING

WARNING

INFO

INFO

FINE

DEBUG1

FINER

DEBUG2

FINEST

DEBUG3

See [Thoughts on logging] for likely future directions in this area.

Mapping an SQL type to a Java class

Using PL/Java, you can install a mapping between an arbitrary type and a Java class. There are two prerequisites for doing this:

- You must know the storage layout of the SQL type that you are mapping.

- The Java class that you map to must implement the interface java.sql.SQLData.

Mapping an existing SQL data type to a java class

Here is an example of how to map the PostgreSQL geometric point type to a Java class. We know that the point is stored as two float8's, the x and the y coordinate.

You can consult the postgresql source code when the exact layout of a basic type is unknown. I peeked at the point_recv function in file src/backend/utils/adt/geo_ops.c to determine the exact layout of the point type.

Once the layout is known, you can create the java.sql.SQLData implementation that uses the class java.sql.SQLInput to read and the class java.sql.SQLOutput to write data:

```
package org.postgresql.pljava.example;

import java.sql.SQLData;
import java.sql.SQLException;
import java.sql.SQLInput;
import java.sql.SQLOutput;

public class Point implements SQLData {
    private double m_x;
    private double m_y;
    private String m_typeName;

    public String getSQLTypeName() {
        return m_typeName;
    }

    public void readSQL(SQLInput stream, String typeName) throws SQLException {
        m_x = stream.readDouble();
        m_y = stream.readDouble();
        m_typeName = typeName;
    }

    public void writeSQL(SQLOutput stream) throws SQLException {
        stream.writeDouble(m_x);
        stream.writeDouble(m_y);
    }

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

```
    */
}
```

Finally, you install the type mapping using the add_type_mapping command:

```
SELECT sqlj.add_type_mapping('point', 'org.postgresql.pljava.example.Point');
```

You should now be able to use your new class. PL/Java will henceforth map any point parameter to the org.postgresql.pljava.example.Point class.

Creating a composite UDT and mapping it to a java class

Here is an example of a complex type created as a composite UDT.

```
CREATE TYPE javatest.complextuple AS (x float8, y float8);

SELECT sqlj.add_type_mapping('javatest.complextuple',
       'org.postgresql.pljava.example.ComplexTuple');

package org.postgresql.pljava.example;

import java.sql.SQLData;
import java.sql.SQLException;
import java.sql.SQLInput;
import java.sql.SQLOutput;

public class ComplexTuple implements SQLData {
   private double m_x;
   private double m_y;
   private String m_typeName;

   public String getSQLTypeName()
   {
      return m_typeName;
   }

   public void readSQL(SQLInput stream, String typeName) throws SQLException
   {
      m_typeName = typeName;
      m_x = stream.readDouble();
      m_y = stream.readDouble();
   }

   public void writeSQL(SQLOutput stream) throws SQLException
   {
      stream.writeDouble(m_x);
      stream.writeDouble(m_y);
```

```
    }

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

Generating SQL automatically

The SQL shown above for this example will be written for you by the Java compiler, if the ComplexTuple class is simply annotated as a "mapped user-defined type" with the desired SQL name and structure:

```
@MappedUDT(schema="javatest", name="complextuple",
    structure={"x float8", "y float8"})
public class ComplexTuple implements SQLData {
  ...
```

Generating the SQL reduces the burden of keeping the definitions in sync in two places. See the hello world example for more.

Packaging tips

This wiki page can be used to gather issues and tips that pertain to building PL/Java packages for downstream distributions or repositories, in between updates to the packaging section in the versioned documentation.

Anyone producing a prebuilt PL/Java package is encouraged to announce its availability on the [[Prebuilt packages]] wiki page.

Parallel query and PL/Java

PL/Java 1.5.1 adds support for PostgreSQL 9.6, and with that comes the possibility of using PL/Java functions in parallel queries. Simple testing shows that this actually works; PL/Java functions can even be declared PARALLEL SAFE if they meet the requirements, and executed in the parallelized parts of queries.

However, this is a substantial change to conditions in which PL/Java was developed, so this wiki page is here to collect the notes that are likely to come with experience using this new capability. Such experience might include empirically-determined, good values for parallel_setup_cost, nonobvious cases where a function should not be declared RESTRICTED or SAFE, and so on.

Notes go here

Preview of new documentation

Until PL/Java 1.5.1 is released, here is a preview of the new section of the user's guide.

PL/Java in parallel query or background worker

With some restrictions, PL/Java can be used in parallel queries, from PostgreSQL 9.6, and in some background worker processes (as introduced in PostgreSQL 9.3, though 9.5 or later is needed for support in PL/Java).

Background worker processes

Because PL/Java requires access to a database containing the sqlj schema, PL/Java is only usable in a worker process that initializes a database connection, which must happen before the first use of any function that depends on PL/Java.

Parallel queries

Like any user-defined function, a PL/Java function can be annotated with a level of "parallel safety", UNSAFE by default.

When a function labeled UNSAFE is used in a query, the query cannot be parallelized at all. If a query contains a function labeled RESTRICTED, parts of the query may execute in parallel, but the part that calls the RESTRICTED function will be executed only in the lead process. A function labeled SAFE may be executed in every process participating in the query.

Parallel setup cost

PostgreSQL parallel query processing uses multiple operating-system processes, and these processes are new for each parallel query. If a PL/Java function is labeled PARALLEL SAFE and is pushed by the query planner to run in the parallel worker processes, each new process will start a Java virtual machine. The cost of doing so will reduce the expected advantage of parallel execution.

To inform the query planner of this trade-off, the value of the PostgreSQL configuration variable parallel_setup_cost should be increased. The startup cost can be minimized with attention to the PL/Java VM option recommendations, including class data sharing.

Limits on `RESTRICTED/SAFE` function behavior

There are stringent limits on what a function labeled RESTRICTED may do, and even more stringent limits on what may be done in a function labeled SAFE. The PostgreSQL manual describes the limits in the section Parallel Labeling for Functions and Aggregates.

While PostgreSQL does check for some inappropriate operations from a PARALLEL SAFE or RESTRICTED function, for the most part it relies on functions being labeled correctly. When in doubt, the conservative approach is to label a function UNSAFE, which can't go wrong. A function mistakenly labeled RESTRICTED or SAFE could produce unpredictable results.

Internal workings of PL/Java While a given PL/Java function itself may clearly qualify as RESTRICTED or SAFE by inspection, there may still be cases where a forbidden operation results from the internal workings of PL/Java itself. This has not been seen in testing (simple parallel queries with RESTRICTED or SAFE PL/Java functions work fine), but to rule out the possibility would require a careful audit of PL/Java's code. Until then, it would be prudent for any application involving parallel query with RESTRICTED or SAFE PL/Java functions to be first tested in a non-production environment.

Further reading

README.parallel in the PostgreSQL source, for more detail on why parallel query works the way it does.

Tuning PL/Java performance

As of 2018, there is a strong selection of Java runtimes that can be used to back PL/Java, including at least:

- Oracle's Java (and Hotspot JVM)
- OpenJDK (with Hotspot JVM)
- OpenJDK (with Eclipse OpenJ9 JVM)

These JVMs offer a wide variety of configurable options affecting both memory footprint and time performance of applications using PL/Java. The options include initial and limit sizes for different memory regions, aggressiveness of just-in-time and ahead-of-time compilation, choice of garbage-collection algorithm, and various forms of shared-memory caching of precompiled classes.

The formal PL/Java documentation contains a fairly extensive treatment of useful Hotspot settings, including a section on plausible minimum settings for memory footprint achievable with different class-sharing and garbage-collector settings. The documentation there of the comparable options and limits for OpenJ9 is more sparse at present.

This wiki page is intended as a clearinghouse for tuning tips and performance measurements for various PL/Java workloads and the available Java runtimes, that can be updated more actively between releases of the formal documentation.

Tip for quickly comparing runtime configurations

Once the PL/Java extension is installed in a database, in any newly-created session, the Java virtual machine is started on the first use of a PL/Java function. The JVM that is started, and how, are determined by the settings of pljava.* configuration variables in effect at that moment, most importantly:

- pljava.libjvm_location selects which Java runtime will be used
- pljava.vmoptions supplies the options to be passed to it

Therefore, all without exiting psql, a new Java runtime or combination of options can be tested by switching to a new connection with \c, setting those options differently, and again calling the PL/Java function of interest.

It can be convenient to include the settings on the psql \c line. For example, to time functionOfInterest() on two different Java runtimes:

```
\c "dbname=postgres options='-c pljava.libjvm_location=/path/to/oracle/.../libjvm
EXPLAIN ANALYZE SELECT functionOfInterest();
\c "dbname=postgres options='-c pljava.libjvm_location=/path/to/openj9/.../libjvm
EXPLAIN ANALYZE SELECT functionOfInterest();
```

For obvious reasons, the pljava.libjvm_location and pljava.vmoptions variables require privilege to set, so the connection needs to be made with superuser credentials.

Sample workload: Java XML manipulation

We will create a table containing a single XML document:

```
CREATE TABLE catalog_as_xml AS
SELECT schema_to_xml('pg_catalog', true, false, '') AS x;
```

In PostgreSQL 11beta3, the resulting document has the following size (after PL/Java and the example code have been loaded):

```
SELECT octet_length(xml_send(x)) AS uncompressed, pg_column_size(x) AS toasted
FROM catalog_as_xml;
```

uncompressed	toasted
14049808	1130828

A test query will return the string value of every element whose string value is exactly six characters (a query that may be artificial and contrived, but can be expressed nearly identically in XML Query (the standard-mandated language for SQL XMLTABLE) and in the PostgreSQL native XMLTABLE syntax, which is limited to XPath 1.0).

The baseline will be the query expressed in XPath 1.0 using the PostgreSQL XMLTABLE function:

```
EXPLAIN ANALYZE SELECT
  xmltable.*
FROM
  catalog_as_xml,
  XMLTABLE('//*[string-length(.) = 6]'
       PASSING x
       COLUMNS s text PATH 'string(.)'
      );
```

It will be compared to the equivalent query expressed in XQuery 1.0 and the "xmltable" function defined in the not-built-by-default org.postgresql.pljava.example.saxon.S9 example, relying on the Saxon-HE library:

```
EXPLAIN ANALYZE SELECT
  xmltable.*
FROM
  catalog_as_xml,
  LATERAL (SELECT x AS ".") AS p,
  "xmltable"('//*[string-length(.) eq 6]',
          PASSING => p,
          COLUMNS => array[ 'string(.)' ]
         )          AS (  s text      );
```

The Java query will be run in both Oracle Java 8 (on the Hotspot JVM) and OpenJDK 8 (with the OpenJ9 JVM), with different choices of class-sharing options:

tag	description
pg	Baseline, PostgreSQL XMLTABLE
hs	Hotspot, no sharing
hs−cds	Hotspot, class data sharing (Java runtime classes only)
hs−appcds	Hotspot, AppCDS (commercial feature), Java runtime, PL/Java, Saxon
j9	OpenJ9, no −Xquickstart, no sharing
j9q	OpenJ9, −Xquickstart, no sharing
j9s	OpenJ9, no −Xquickstart, sharing (Java runtime, PL/Java, Saxon)
j9qs	OpenJ9, −Xquickstart, sharing (as above)

EXPLAIN ANALYZE reported timings in milliseconds:

iteration	pg	hs	hs−cds	hs−appcds	j9	j9q	j9s	j9qs
1st	908.231	1888.859	1837.186	1539.781	3250.965	3095.733	2443.649	2644.991
2nd	879.483	772.545	838.082	826.558	1229.200	1855.513	1073.335	1932.083
4th	881.302	664.422	688.487	673.037	1011.018	1708.208	987.191	1912.010
8th	880.766	640.940	643.535	632.260	962.517	1660.867	952.857	1870.506
16th	880.622	654.674	682.772	627.037	967.805	1656.651	943.923	1941.888

Discussion

- The baseline native XMLTABLE implementation in PostgreSQL delivers consistent times over successive runs. Java timings improve over successive early runs, as the VM identifies and reoptimizes hot areas.
- For all of the Java results, the first-iteration result includes the time to launch the Java virtual machine. For Hotspot, this gives a time to first result from 67% (best) to 108% (worst) longer than the native baseline.
- All tested Hotspot configurations are outperforming the native implementation as soon as the next iteration, and eventually by 22% to 28%.
- For this workload, Hotspot seems to have a striking performance advantage relative to OpenJ9. Possible explanations:
  - Saxon is a mature and carefully-optimized library; are its optimizations extremely specific to Hotspot?
  - PL/Java makes heavy use of JNI; could this pattern be less well handled in OpenJ9?
- OpenJ9's −Xquickstart is a poor fit for this workload, as it suppresses JIT optimization so drastically that performance improves very little on successive runs.
- The combination of −Xquickstart and −Xshareclasses for this workload is especially disappointing, probably because the two features, when combined, force the ahead-of-time compilation of all methods. That sounds promising, but not if the AOT code significantly underperforms what the optimizing JIT would generate.
- Memory footprint was not compared. PL/Java's documentation already has a section on plausible memory settings for Hotspot, but not for OpenJ9, which has a good reputation for memory frugality. Exploration would be worthwhile.
- There could be other workloads in which the Hotspot and OpenJ9 relative timings could be closer, or even reversed.
- The procedure to set up class sharing for OpenJ9 is considerably simpler than to set up AppCDS for Hotspot, enough to make OpenJ9 an attractive choice for workloads where the performance is more comparable.

Variation by processor count

The results above were obtained with 6 available processor cores (12 hyperthreads). Here, the best Hotspot (h−) and OpenJ9 (j−) configurations from above (hs−appcds and j9s, respectively) are repeated for different numbers of cores and threads available to the backend process.

iteration	h−4c8t	h−4c4t	h−2c4t	h−2c2t	h−1c2t	h−1c1t
1st	1798.020	2140.068	1871.740	2760.872	2564.169	4306.058
2nd	780.182	827.379	825.943	1054.749	1064.300	1704.112
4th	661.740	672.415	662.259	786.407	734.421	756.054
8th	619.978	653.686	641.784	659.112	678.855	722.792
16th	619.824	647.092	664.287	671.862	639.365	651.502

iteration	j−4c8t	j−4c4t	j−2c4t	j−2c2t	j−1c2t	j−1c1t
1st	2413.365	2419.176	2411.006	2470.092	2457.868	3673.986
2nd	1108.991	1093.504	1050.731	1142.424	1072.635	2599.973
4th	969.465	1003.736	988.883	983.431	941.495	1032.896
8th	967.447	900.644	963.011	926.342	920.390	1032.316
16th	1113.963	932.503	1496.240	925.137	939.179	990.072

Discussion Hotspot's initial startup uses parallelism to good advantage, so the startup time suffers when cores are limited, and especially when limited to one hardware thread. Interestingly, comparing sets with the same number of threads, in one case independent on an equal number of cores, and in the other case hyperthreads on half as many cores, the data above seem to favor the hyperthreaded case. More runs might reveal whether that apparent pattern persists.

Even with only one hardware thread available, Hotspot can still produce code that outperforms the native libxml2 no later than the fourth iteration.

OpenJ9, while not achieving the ultimate speeds of Hotspot on this workload, shows a first-run time that suffers less when limited to few CPUs. However, that advantage diminishes when taking the times of the first two runs into account.

Not shown in these tables, but as expected, the baseline PostgreSQL native XMLTABLE posted timings of 893 ms first run, 877 ms second run, consistently with the earlier values, even when limited to one core, one thread.

Notes on methodology

Platform Intel Xeon X5650 2.67 GHz, 6 cores (12 hyperthreads), 24 GB RAM, Linux.

PostgreSQL installation, Java runtimes, database, and PL/Java and Saxon libraries and jars installed in an in-memory (tmpfs) filesystem.

Connection strings used for each test configuration *Note: the connection strings below for the Hotspot runs with AppCDS contain the option −XX:+UnlockCommercialFeatures because the runs were done on Oracle Java 8 where AppCDS is a commercial feature, and its use in production will need a license from Oracle. The same feature appears in OpenJDK with Hotspot starting in Java 10, where it is not a commercial feature, and does not require that −XX:+UnlockCommercialFeatures option; it is otherwise configured in the same way.*

```
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/nohome/jre/lib/a
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/nohome/jre/lib/a
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/nohome/jre/lib/a
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/jdk8u162-b12_ope
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/jdk8u162-b12_ope
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/jdk8u162-b12_ope
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/jdk8u162-b12_ope
```

Jars loaded into PL/Java The PL/Java sqlj.install_jar function was used to install the PL/Java examples jar (giving it the name ex), with deploy => true to create the function declarations, and also the Saxon−HE−9.8.0−14.jar, naming it saxon.

The PL/Java application classpath (set with sqlj.set_classpath on the public schema), was ex during the Hotspot runs, and ex:saxon during the OpenJ9 runs. (For the Hotspot runs, the Saxon jar was placed on the system classpath by adding it to pljava.classpath instead, as explained below.)

Setup for Hotspot

- The existing Hotspot installation on disk was copied to the tmpfs.
- That invalidates the paths in the supplied classes.jsa shared archive that was generated when Java was installed to its location on disk, so the lib/amd64/server/classes.jsa file was removed from the copy and regenerated with java −Xshare:dump to contain the correct paths. That shared archive contains only classes of the Java runtime itself.
- The shared archive for AppCDS, to include PL/Java implementation classes and the Saxon library as well as the Java runtime's classes, was generated in two steps:
  1. A connection string with −XX:DumpLoadedClassList=filename was issued and the test query was executed, to populate the class list with the needed classes.
  2. A new connection string with −Xshare:dump and −XX:SharedClassListFile naming the classlist file generated in the first step was issued, and

then SELECT sqlj.get_classpath('public'); to trigger PL/Java loading. Java reads the class list and generates the shared archive, and the backend exits.

- Because Hotspot AppCDS will share only classes from the system classpath, the pljava.classpath setting was altered to include Saxon−HE−9.8.0−14.jar as well as the PL/Java jar.
- Because PL/Java's security manager disallows jar loading from arbitrary filesystem locations, the Saxon−HE−9.8.0−14.jar was placed in Java's jre/lib directory and the pljava.classpath referred to it there.
- AppCDS will not share classes contained in a signed jar, and the distributed Saxon−HE−9.8.0−14.jar is signed, so the copy placed in jre/lib was "de-signed" by deleting its TE−050AC.SF entry and all Name:/Digest: sections from its MANIFEST.MF entry.

Setup for OpenJ9

- The OpenJDK with OpenJ9 download was unzipped in the /var/tmp tmpfs.
- Because PL/Java under OpenJ9 is able to share classes from the PL/-Java application classpath (the one managed by sqlj.set_classpath) and not just the system classpath, there was no need to add the Saxon jar to pljava.classpath as there was for Hotspot. It was simply loaded with sqlj.install_jar under the name saxon, and put on the application classpath with SELECT sqlj.set_classpath('public', 'ex:saxon');.
- Each set of runs with sharing (j9s, j9qs) was prepared by starting a fresh session with the same connection string to be used for that set, and the shareDir named in that connection string empty. Sixteen runs were made without timing, to populate the shared cache.
- Then the same connection string was used again to start a fresh session, and the full set of 16 runs repeated and timed.

Connection strings generating `AppCDS` shared archive *See the earlier note concerning the −XX:+UnlockCommercialFeatures option, which is needed (with legal implications) to use the AppCDS feature in Oracle Java. The same feature appears in OpenJDK as of Java 10, without the need for that option or a commercial license.*

```
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/nohome/jre/lib/a
\c "dbname=postgres options='-c pljava.libjvm_location=/var/tmp/nohome/jre/lib/a
```

"De-signing" the Saxon jar Hotspot's AppCDS will not share classes from a signed jar, so the signatures were removed from the Saxon jar with this procedure:

```
zip -d Saxon-HE-9.8.0-14.jar META-INF/TE-050AC.SF
unzip Saxon-HE-9.8.0-14.jar META-INF/MANIFEST.MF
```

```
ed META-INF/MANIFEST.MF <<END-COMMANDS
/^[[:space:]]/+1,$d
wq
END-COMMANDS
zip -u Saxon-HE-9.8.0-14.jar META-INF/MANIFEST.MF
```

Stripping the signatures does not impair the operation of the open-source Saxon-HE. It is conceivable that the commercial Saxon-PE or Saxon-EE would object to such treatment.

Setup for processor-count variation Several Linux control groups were created as follows:

```
mkdir /sys/fs/cgroup/cpuset/{1c1t,1c2t,2c2t,2c4t,4c4t,4c8t}
for i in /sys/fs/cgroup/cpuset/?c?t
do
  echo 0 >$i/cpuset.mems
done
echo 0       >/sys/fs/cgroup/cpuset/1c1t/cpuset.cpus
echo 0,1     >/sys/fs/cgroup/cpuset/1c2t/cpuset.cpus
echo 0,2     >/sys/fs/cgroup/cpuset/2c2t/cpuset.cpus
echo 0-3     >/sys/fs/cgroup/cpuset/2c4t/cpuset.cpus
echo 0,2,4,6 >/sys/fs/cgroup/cpuset/4c4t/cpuset.cpus
echo 0-7     >/sys/fs/cgroup/cpuset/4c8t/cpuset.cpus
```

After each new backend was established with the appropriate \c line, its process ID was obtained with SELECT pg_backend_pid(); and echoed into cgroup.procs in the appropriate cpuset subdirectory.

The OpenJ9 class share was initially populated with one set of 16 runs before any timing was done. Timings were then done in the order shown, from 4c8t to 1c1t, and the -Xshareclasses option did not have readonly added for the timed sets. Because OpenJ9 can continue adding JIT hints to a class share during operation, it is possible that the later sets benefit from JIT hints added during the earlier ones.

- Neither the name Tada AB nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Prebuilt PL/Java distributions

At present, the PL/Java project is reliant on downstream packagers to produce prebuilt, installable PL/Java packages for various platforms. The official PL/-Java releases are offered in source form and take only a few minutes to build with Apache Maven as described in the build instructions.

This wiki page will be updated to list known prebuilt PL/Java packages and the platforms they are built for. As with any prebuilt distribution, you should be acquainted with the policies and reputation of any supplier of a prebuilt package. The PL/Java project has not directly built or verified any package listed here.

Known prebuilt packages available

Debian/Ubuntu packages on apt.postgresql.org

```
{1.5.2,"11.1 (Debian 11.1-1.pgdg+1)",11.0.1,Linux,amd64}‘
```

PL/Java 1.5.2 packages available for PostgreSQL 11 back to 9.3 for Debian unstable/buster/stretch and Ubuntu cosmic/bionic/xenial, for amd64/i386/ppc64el. Dbgsym packages available. Includes pljava-examples jar with the optional Saxon examples already built (download Saxon-HE 9.8.0.14 jar separately to use them).

Added 14 November 2018

Docker images

Martin Bednar has prepared images of 64-bit PostgreSQL (9.5 and 9.4) with PL/Java 1.5.0 and Oracle Java 8 for use with Docker.

```
{1.5.0,9.5.1,1.8.0_74,Linux,amd64}
{1.5.0,9.4.6,1.8.0_74,Linux,amd64}
```

added 12 April 2016

Complete PostgreSQL distributions from BigSQL

BigSQL provides native installers for Centos 6 and 7, Ubuntu 12.04 and 14.04, OS X 10.9+, Windows 7+, and Windows Server 2008 and 2012. These distributions of PostgreSQL 9.5, 9.4, 9.3, and 9.2 include PL/Java 1.5.0.

added 12 April 2016

To list a prebuilt package here

Please announce the availability of your package on the pljava-dev mailing list, along with the output of the third query below:

Note: as of mid-May 2016, the pljava—dev mailing list is working again, and should be used to announce packages. In case the mailing list does not seem to work, then please open an issue.

```
SELECT sqlj.install_jar( fileurl-to-built-pljava-examples-*.jar , 'ex', true);
SELECT sqlj.set_classpath('javatest', 'ex');
```

```
SELECT array_agg(java_getsystemproperty(p)) FROM (values
('org.postgresql.pljava.version'),
('org.postgresql.version'),
('java.version'),
('os.name'),
('os.arch')
) AS props(p);
```

2017/06 Breaking news: workaround crash with Stack Guard-hardened kernels

As of late June 2017, Linux kernel vendors are shipping updates that harden the kernel against certain so-called stack-smash attacks. The hardened kernels change the mapping of memory just below the stack, and cause Java to crash with a SIGBUS error (as reported elsewhere, not only in PL/Java). If you experience this, add −Xss2M (or larger than 2M, if a larger stack size is needed by your application) to pljava.vmoptions.

For more information, see PL/Java issue #129 and (for Red Hat subscribers) this Red Hat solutions document.

Figure 1: PL/Java

Welcome to PL/Java

If you have comments or ideas regarding this wiki, please convey them on the Mailing List. A great deal of information can also be found at the project information site.

Overview

PL/Java is a free add-on module that brings Java™ Stored Procedures, Triggers, and Functions to the PostgreSQL™ backend. The development started late 2003 and the first release of PL/Java arrived in January 2005. The project is released under the [[PLJava License]] license.

Features

- Ability to write functions, triggers, user-defined types, … using recent Java versions. (To see the currently-targeted versions, please see the versions page.)
- Standardized utilities (modeled after the SQL 2003 proposal) to install and maintain Java code in the database.
- Standardized mappings of parameters and result. Supports scalar and composite user-defined types (UDTs), pseudo types, arrays, and sets.
- An embedded, high performance JDBC driver utilizing the internal PostgreSQL SPI routines.
- Metadata support for the JDBC driver. Both DatabaseMetaData and ResultSetMetaData are included.
- Integration with PostgreSQL savepoints and exception handling.
- Ability to use IN, INOUT, and OUT parameters
- Two language handlers, javau (functions not restricted in behavior, only superusers can create them) and java (functions run under a security man-

ager blocking filesystem access, users who can create them configurable with GRANT/REVOKE)
- Transaction and Savepoint listeners enabling code execution when a transaction or savepoint is commited or rolled back.

PL/Java earlier supported GCJ, but targets conventional Java virtual machines for current development.

Documentation

The first stop for *up-to-date* documentation should be the project information site.

You may also find useful information via the wiki links below. Information here will be migrating to the project information site as it is brought up to date.

[[Installation Guide]]
[[User Guide]]
[[Contribution Guide]]

Resources

Note: To be sure of running a current PL/Java, please check the releases page to see what is current. You may check for any [[prebuilt packages]] available for your platform. If prebuilt packages are not available for your platform, or if they are behind the current version, please consider Building PL/Java using the source here on GitHub.

The "no longer supported" downloads linked below are quite old and of chiefly historical interest.

Source downloads
[[Prebuilt packages]]
No longer supported downloads
Mailing List
Questions tagged pljava on Stack Overflow (Atom feed)
Feed for changes to this wiki itself
Bug Tracking
Older bug tracker at PgFoundry
Even older bug tracker at GBorg

Technology

[[Technology in Brief]]
[[The choice of JNI]]

Returning complex types

The SQL-2003 draft suggest that a complex return value is handled as an IN-/OUT parameter and PL/Java implements it that way. If you declare a function that returns a complex type, you will need to use a Java method with boolean return type with a last parameter of type java.sql.ResultSet added after all of the method's visible parameters. The output parameter will be initialized to an updatable ResultSet that contains exactly one row.

```
CREATE FUNCTION createComplexTest(int, int)
  RETURNS complexTest
  AS 'foo.fee.Fum.createComplexTest'
  IMMUTABLE LANGUAGE java;
```

The PL/Java method resolver will now find the following method in class foo.fee.Fum:

```
public static boolean complexReturn(int base, int increment, ResultSet receiver)
throws SQLException
{
  receiver.updateInt(1, base);
  receiver.updateInt(2, base + increment);
  receiver.updateTimestamp(3, new Timestamp(System.currentTimeMillis()));
  return true;
}
```

The return value denotes if the receiver should be considered as a valid tuple (true) or NULL (false).

Running PL/Java sample tests

The PL/Java Source distribution contains a couple of rudimentary tests. The tests are divided into two jar files. One is the client part found in the test.jar. It contains some methods that executes SQL statements and prints the output (all contained there can of course also be executed from psql or any other client). The other is the examples.jar which contains the sample code that runs in the backend. The latter must be installed in the database in order to function. An easy way to do this is to use psql and issue the command:

```
SELECT sqlj.install_jar('file:///some/directory/examples.jar', 'samples',
true);
```

Please note that the deployment descriptor stored in examples.jar will attempt to create the schema javatest so the user that executes the sqlj.install_jar command must have permission to do that. A number of tests now run from the deployment descriptor itself, so by the time install_jar finishes, PL/Java will have completed those tests.

Once loaded, you must also set the classpath used by the PL/Java runtime. This classpath is set per schema (namespace). A schema that lacks a classpath will default to the classpath that has been set for the public schema. The tests will use the schema javatest. To define the classpath for this schema, simply use psql and issue the command:

```
SELECT sqlj.set_classpath('javatest', 'samples');
```

The first argument is the name of the schema, the second is a colon separated list of jar names. The names must reflect jars that are installed in the system.

NOTE: If you don't use schemas, you must still issue the set_classpath command to assign a correct classpath to the 'public' schema. This can only be done by a super user.

Now, you should be able to run the client test application:

```
java -cp <path including the jdbc driver and test.jar> org.postgresql.pljava.tes
```

Savepoints

PostgreSQL savepoints are exposed using the standard setSavepoint() and releaseSavepoint() methods on the java.sql.Connection interface. Two restrictions apply:

- A savepoint must be rolled back or released in the function where it was set.
- A savepoint must not outlive the function where it was set.

"Function" here refers to the PL/Java function that is called from SQL. The restrictions do not prevent the Java code from being organized into several methods, but the savepoint cannot survive the eventual return from Java to the SQL caller.

Installation

Only a PostgreSQL super user can install PL/Java. The PL/Java utility functions are installed as "security definer" so that they execute with the access permissions that were granted to the creator of the functions.

Trusted vs. untrusted language

PL/Java can declare two language entries in SQL: java and javau. Following the conventions of other PostgreSQL PLs, the 'untrusted' language (javau) places no restrictions on what the Java code can do, while the 'trusted' language (java) installs a security manager that restricts access to the filesystem.

GRANT/REVOKE USAGE ON LANGUAGE java can be used to regulate which users are able to create functions in the java language. For the javau language, regardless of permissions, only superusers can create functions.

Execution of the deployment descriptor

The install_jar, replace-jar, and remove_jar utility functions optionally execute commands found in a [[SQL deployment descriptor]]. Such commands are executed with the permissions of the caller. In other words, although the utility function is declared with "security definer", it switches back to the identity of the invoker during execution of the deployment descriptor commands.

Classpath manipulation

The utility function set_classpath requires that the caller of the function has been granted *CREATE* permission on the affected schema, unless it is the public schema, in which case the caller must be a superuser.

##Linux threads cause sporadic hanging##

It seems Java doesn't play nice with LinuxThreads. I rebuilt glibc to use the Native POSIX Thread Library (NPTL) and restarted PostgreSQL. Everything seems to be working so far. Here's how you can check what you have:

$ getconf GNU_LIBPTHREAD_VERSION
linuxthreads-0.10

If you see linuxthreads, you need to upgrade. This is what you want to see:

$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.3.6

(version might be higher) You can also get this information (and more) by running /libc.so.6:

$ /lib/libc.so.6
...
linuxthreads-0.10 by Xavier Leroy
...

or:

$ /lib/libc.so.6
...
Native POSIX Threads Library by Ulrich Drepper et al
...

If you can't switch to NPTL for some reason, it might be possible to use LD_ASSUME_KERNEL to get things working on LinuxThreads.

###References### http://docs.oracle.com/cd/E13924_01/coh.340/cohfaq/faq16702.htm
http://en.wikipedia.org/wiki/NPTL http://gentoo-wiki.com/NPTL#Switching_to_NPTL
http://people.redhat.com/drepper/assumekernel.html http://developer.novell.com/wiki/index.php/LD_ASSU

SQLJ deployment descriptors

The install_jar, replace_jar, and remove_jar functions can act on a *deployment descriptor* allowing SQL commands to be executed after the jar has been installed or prior to removal.

The descriptor is added as a normal text file to your jar file. In the Manifest of the jar there must be an entry that appoints the file as the SQLJ deployment descriptor.

```
Name: deployment/examples.ddr
SQLJDeploymentDescriptor: TRUE
```

Such a file can be written by hand according to the format below, but the usual method is to add specific Java annotations in the source code, as described under function mapping - SQL generation. The Java compiler then generates the deployment descriptor file at the same time it compiles the Java sources, and the compiled classes and .ddr file can all be placed in the jar together.

The format of the deployment descriptor is stipulated by ISO/IEC 9075-13:2003.

```
<descriptor file> ::=
  SQLActions <left bracket> <rightbracket> <equal sign>
  { [ <double quote> <action group> <double quote>
    [ <comma> <double quote> <action group> <double quote> ] ] }

<action group> ::=
    <install actions>
  | <remove actions>

<install actions> ::=
  BEGIN INSTALL [ <command> <semicolon> ]... END INSTALL

<remove actions> ::=
  BEGIN REMOVE [ <command> <semicolon> ]... END REMOVE

<command> ::=
    <SQL statement>
  | <implementor block>

<SQL statement> ::= <SQL token>...

<implementor block> ::=
```

```
  BEGIN <implementor name> <SQL token>... END <implementor name>

<implementor name> ::= <identifier>

<SQL token> ::= ! an SQL lexical unit specified by the term "<token>"
                  in Sub clause 5.2, "<token> and <separator>", in ISO/IEC 9075-
```

If implementor blocks are used, PL/Java will consider only those with implementor name PostgreSQL (case insensitive) by default. Here is a sample deployment descriptor:

```
SQLActions[] = {
  "BEGIN INSTALL
    CREATE FUNCTION javatest.java_getTimestamp()
      RETURNS timestamp
      AS 'org.postgresql.pljava.example.Parameters.getTimestamp'
      LANGUAGE java;
      END INSTALL",
  "BEGIN REMOVE
    DROP FUNCTION javatest.java_getTimestamp();
  END REMOVE"
}
```

Configurable implementor-block recognition

Although, by default, only the implementor name PostgreSQL is recognized, the implementor name(s) to be recognized can be set as a list in the variable pljava.implementors. It is consulted after every command while executing a deployment descriptor, which gives code in the descriptor a rudimentary form of conditional execution control, by changing which implementor blocks will be executed based on discovered conditions.

Functions in the sqlj schema

install_jar

The install_jar command loads a jarfile from a location appointed by an URL into the SQLJ jar repository. It is an error if a jar with the given name already exists in the repository. #### Usage

SELECT sqlj.install_jar(<jar_url>, <jar_name>, <deploy>);

Parameter

Description

jar_url

The URL that denotes the location of the jar that should be loaded

jar_name

This is the name by which this jar can be referenced once it has been loaded

deploy

True if the jar should be deployed according to a deployment descriptor, false otherwise

replace_jar

The replace_jar command will replace a loaded jar with another jar. Use it to update already loaded files. It's an error if the jar is not found.

Usage SELECT sqlj.replace_jar(<jar_url>, <jar_name>, <redeploy>);

Parameter

Description

jar_url

The URL that denotes the location of the jar that should be loaded.

jar_name

The name of the jar to be replaced.

redeploy

True if the jar should be undeployed according to the deployment descriptor of the old jar and deployed according to the deployment descriptor of the new jar, false otherwise.

remove_jar

The remove_jar command will drop the jar from the jar repository. Any classpath that references this jar will be updated accordingly. It's an error if the jar is not found.

Usage SELECT sqlj.remove_jar(<jar_name>, <undeploy>);

Parameter

Description

jar_name

The name of the jar to be removed.

undeploy

True if the jar should be undeployed according to a deployment descriptor, false otherwise.

get_classpath

The get_classpath command will return the classpath that has been defined for the given schema. NULL is returned if the schema has no classpath. It's an error if the given schema does not exist.

Usage SELECT sqlj.get_classpath(<schema>);

Parameter

Description

schema

The name of the schema

set_classpath

The set_classpath command will define a classpath for the given schema. A classpath consists of a colon separated list of jar names. It's an error if the given schema does not exist or if one or more jar names references nonexistent jars.

Usage SELECT sqlj.set_classpath(<schema>, <classpath>);

Parameter

Description

schema

The name of the schema.

classpath

The colon separated list of jar names.

add_type_mapping

The add_type_mapping command installs a mapping between a SQL type and a Java class. Once the mapping is in place, parameters and return values will be mapped accordingly. Please read [Mapping an SQL type to a Java class] for detailed information.

Usage SELECT sqlj.add_type_mapping(<sql type>, <java class>);

Parameter

Description

sql type

The name of the SQL type. The name can be qualified with a schema (namespace). If the schema is omitted, it will be resolved according to the current setting of the search_path.

java class

The name of the class. The class must be found in the classpath in effect for the current schema

drop_type_mapping

The drop_type_mapping command removes a mapping between a SQL type and a Java class.

Usage SELECT sqlj.drop_type_mapping(<sql type>);

Parameter

Description

sql type

The name of the SQL type. The name can be qualified with a schema (namespace). If the schema is omitted, it will be resolved according to the current setting of the search_path.

Note on jar URLs

The install_jar and replace_jar commands accept a URL (that must be reachable from the server) to a jar file. It is even possible, using the rules for jar URLs, to construct one that refers to a jar file within another jar file. For example:

```
jar:file:outer.jar!/inner.jar
```

However, Java's caching of the "outer" jar may frustrate attempts to replace or reload a newer version within the same session.

A function or trigger in SQL resolves to a static method in a Java class. In order for the function to execute, the appointed class must be installed in the database. PL/Java adds a set of functions that helps installing and maintaining the java classes. Classes are loaded into the database from normal Java archives (AKA jars). A Jar may optionally contain a deployment descriptor that in turn contains SQL commands to be executed when the jar is deployed/undeployed. The functions are modeled after the standards proposed for SQL 2003.

PL/Java implements a standardized way of passing parameters and return values. Complex types and sets are passed using the standard JDBC ResultSet class. Great care has been taken not to introduce any proprietary interfaces unless absolutely necessary so that Java code written using PL/Java becomes as database agnostic as possible.

A JDBC driver is included in PL/Java. This driver is written directly on top of the PostgreSQL internal SPI routines. This driver is essential since it's very common for functions and triggers to reuse the database. When they do, they must use the same transactional boundaries that where used by the caller.

PL/Java is optimized for performance. The Java virtual machine executes within the same process as the backend itself. This vouches for a very low call overhead. PL/Java is designed with the objective to enable the power of Java to the database itself so that database intensive business logic can execute as close to the actual data as possible.

The standard Java Native Interface (JNI) is used when bridging calls from the backend into the Java VM and vice versa. Please read the rationale behind [[The choice of JNI]] and a more in-depth discussion about some implementation details.

The versions of PostgreSQL and Java targeted by current PL/Java development can be reviewed on the versions page.

Rationale behind using JNI as opposed to threads in a remote JVM process.

Reasons to use a high level language like Java™ in the backend

A large part of the reason why JNI was chosen in favor of an RPC based, single JVM solution was due to the expected use-cases. Enterprise systems today are almost always 3-tier or n-tier. Database functions, triggers, and stored procedures are mechanisms that extend the functionality of the backend tier. They typically rely on a tight integration with the database due to a very high rate of interactions and execute inside of the database largely to limit the number of interactions between the middle tier and the backend tier. Some typical use-cases:

- Referential integrity enforcement. Using Java, referential integrity can be implemented that goes beyond what can be done using the standard SQL semantics. It may involve checking XML documents, enforcing some meta-driven rule system, or other complex tasks that put high demands on the implementation language.
- Advanced pattern recognition. Soundex, image comparison, etc.
- XML support functions. Java comes with a lot of XML support. Parsers etc. are readily available.
- Support functions for O/R mappers. A variety of support can be implemented depending on design. One example is an O/R mapper that allows methods on persistent objects. A lot can be gained if such methods are pushed down and executed within the database. Consider the following (OQL):

```
SELECT AVG(x.salary - x.computeTax()) FROM Employee x WHERE x.salary > 120000;
```

Pushing the computeTax logic down to the database instead of computing it in the middle tier (where much or the O/R logic resides) is a huge gain from a performance standpoint. The statement could be transformed into SQL as:

```
SELECT AVG(x.salary - computeTax(x.salary)) FROM Employee x WHERE x.salary > 120
```

As a result, very few interactions (typically only one) need to be made between the middle and the backend tier.

- Views and indexes making use of computed values. In the above example and index could be created on computeTax(x.salary) and a view could express that as net_income.
- Message queue management. Delivering or fetching things using message queues or other delivery mechanisms. As with most interactions with other processes, this requires transaction coordination of some kind.

One might argue that since a JVM often is present running an app-server in the middle tier, would it not be more efficient if that JVM also executed the database functions and triggers? In my opinion, this would be very bad. One major reason for moving execution down to the database is performance (by minimizing the number of roundtrips between the app-server and the database) another is separation of concern. Referential data integrity and other ways to extend the functionality of the database should not be the app-servers concern, it belongs in the backend tier. Other aspects like database versus app-server administration, replication of code and permission changes for functions, and running different tiers on different servers, makes it even worse.

Resource consumption

Having one JVM per connection instead of one thread per connection running in the same JVM will undoubtedly consume more resources. There are however a couple of facts that must be remembered:

- The overhead of multiple processes is already present due to the fact that each connection is a process in a PostgreSQL system.
- In order to keep connections separated in case they run in the same JVM, some kind of "compartments" must be created. Either you create them using parallel class loader chains (similar to how EAR files are managed in an EJB server) or you use a less protective model similar to a servlet engine. In order to get a separation that is comparable to what you get using separate JVM's, you must chose the former. That consumes some resources.
- The JVM has undergone a series of improvements in order to reduce footprint and startup time. Some significant improvements where made in Java 1.4 and Java 1.5 introduces Java Heap Self Tuning, Class Data Sharing, and Garbage Collector Ergonomics (read more here), technologies that will minimize the startup time and make the JVM adopt its resource consumption in a much improved way.
- PL/Java can make use of the GCJ. Using this technology, all core classes will be compiled into binaries and optionally pre-loaded by the postmaster. It also means that all modules that are loaded using the install_jar/replace_jar can be compiled into real shared objects. Finally, it means that the footprint for each "JVM" will be significantly decreased.

Note: GCJ is no longer targeted by current PL/Java development.

Connection pooling

In the Java community you are very likely to use a connection pool. The pool will ensure that the number of connections stays as low as possible and that connections are reused (instead of closed and reestablished). New JVMs are started rarely.

Connection isolation

Separate JVMs gives you a much higher degree of isolation. This brings a number of advantages:

- There's no problem attaching a debugger to one connection (one JVM) while the others run unaffected.
- There's no chance that one connection manages to accidentally (or maliciously) exchange dirty data with another connection.
- A process that performs tasks that consume a lot of CPU under a long period of time can be scheduled with a lower priority using a simple OS command.
- The JVMs can be brought down and restarted individually.
- Security policies are much easier to enforce.

Transaction visibility

In order to maintain the correct visibility, the transaction must somehow be propagated to the Java layer. I can see two solutions for this using RPC. Either an XA aware JDBC-driver is used (requires XA support from PostgreSQL) or a JDBC driver is written so that it calls back to the SPI functions in the invoking process. Both choices results in an increased number of RPC calls and a negative performance impact.

The PL/Java approach is to use the underlying SPI interfaces directly through JNI by providing a "pseudo connection" that implements the JDBC interfaces. The mapping is thus very direct. Data need never be serialized nor duplicated.

RPC performance

Remote procedure calls are extremely expensive compared to in-process calls. Relying on an RPC mechanism for Java calls will cripple the usefulness of such an implementation a great deal. Here are two examples:

- In order for an update trigger to function using RPC, you can choose one of two approaches. Either you limit the number of RPC calls and send two full Tuples (old and new) and a Tuple Descriptor to the remote JVM, and then pass a third Tuple (the modified new) back to the original, or you pass those structures by reference (as CORBA remote objects) and

perform one RPC call each time you access them. You have a tradeoff between on one hand, limited functionality and poor performance, and on the other, good functionality and really bad performance.

- When one or several Java functions are used in the projection or filter of a SELECT statement on a query processing several thousand rows, each row will cause at least one call to Java. In case of RPC, this implies that the OS needs to do at least two context switches (back and forth) for each row in the query.

Using JNI to directly access structures like TriggerData, Relation, TupleDesc, and HeapTuple minimizes the amount of data that needs to be copied. Parameters and return values that are primitives need not even become Java objects. A 32-bit int4 Datum can be directly passed as a Java int (jint in JNI).

Simplicity

I've have some experience of work involving CORBA and other RPCs. They add a fair amount of complexity to the process. JNI however, is invisible to the user.

Thoughts on logging

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Note: this page does not describe how PL/Java currently works, except in the "Background" part. It is a proposal for further de-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Also, to anyone reading this for review or comment, a lot of this material may be more familiar than it is to

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Also also, to some it may seem strange that I use phrases like "log event" without insisting on an essential difference be-

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

So,
here
goes.
…
*log-
ging
isn't
par-
ticu-
larly
mag-
ical.*

—
Dave
Cramer

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

…
it's
what
you
do
with
it.

—
variously
at-
tributed

##
Back-
ground

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

PostgreSQL has supremely good log messages.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

It even has a [style guide][stygd] for writing them, and it pays off in the well-known quality and helpfulness of PostgreSQL mes-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[stygd]: http://www.postgresql.org/docs/current/static/error-style-guide.html

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Part of the excellence of PostgreSQL's messages can be traced to their rich structure. A message is not a blob of text

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

item
|pq|PL/pgSQL
|pgjdbc
(+
-ng
Notice)|pgjdbc-
ng
Exc
|PL/-
Java
———
———
—-
|—
|—
———
———
——
|—
———
—45
———
|—
———
——
|—
———
———

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The libpq on-the-wire protocol preserves this structure, sending these components (the ones with pq codes) distinctly and in-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

How awesome is that? Consider this: I have seen, with my own eyes, non-technical users entering stuff into a PostgreSQL database (us-

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

I
chal-
lenge
any-
one
who
has
had
sup-
port
ex-
peri-
ence
to
tell
me
that
ain't
magic.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

By and large, messages in PostgreSQL really are *that* good.

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

###
Orig-
inal
log
event
life
cy-
cle

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

A message that originates in the PostgreSQL backend proper begins as a call to ereport or elog in [elog.c][elogc]. The rules there are

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

0. If the message has any severity *below* ERROR (so, DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, COMMERROR, INFO, NOTICE, or WARNING) **or** *above* ERROR (so, FATAL,

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

0. If the severity is *exactly* ERROR, it gets *thrown* PostgreSQL-style, and can be caught in PG_TRY/PG_CATCH constructs. The [elog][elogc] code does *not* send it to

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[elogc]: http://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backend

Handling logged events in front-end code

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Events from the server that arrive at the front end show up in a form the front-end code can inspect and choose how to handle,

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

This code in turn might want to log an event (whether one received from the backend as just described, or one originating in

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

For code that is really running in a front end, that's the end of the story, but for code running in a back-end PL, the story has

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

Handling logged events in a backend PL

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The first requirements for server-side code are the same as for the front end: it should be able to intercept, examine,

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

In PL/pgSQL, all of that applies to events with the exact severity ERROR: warnings/notices/etc. are in visible to PL/pgSQL code, as the

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[plpgscatch]: http://www.postgresql.org/docs/current/static/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING

[plpgsdiag]: http://www.postgresql.org/docs/current/static/plpgsql-control-structures.html#PLPGSQL-EXCEPTION-DIAGNOSTICS

####

As for PL/-Java

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PL/Java is at parity with PL/pgSQL as far as the ability to catch error events from the backend (as Java SQLExceptions), or to propa-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Internally, it has good provisions for examining individual properties of the event (currently not including the

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[fnames]: http://git.postgresql.org/gitweb/?p=postgresql.git;a=blobdiff;f=src/incl
[servx]: http://tada.github.io/pljava/pljava/apidocs/index.html?org/postgresql/
[edata]: http://tada.github.io/pljava/pljava/apidocs/index.html?org/postgresql/

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

(In passing, the current design where the magic only happens for a single ServerException subclass of SQLException stands in the way of im-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[catex]: http://www.javaspecialists.eu/archive/Issue138.html

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

This is one area where a good API should be worked out and published. PL/-Java provides a JDBC interface to the backend,

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

(Again in passing, the fact that PL/-Java presents a JDBC interface could raise hopes that PL/-Java functions are also able to ex-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[unwrap]: http://docs.oracle.com/javase/7/docs/api/index.html?java/sql/Wrapper
[jdbcw]: http://docs.oracle.com/javase/7/docs/api/index.html?java/sql/ResultSe

---

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

How do the other PostgreSQL JDBCs give access to error details?

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

#####
pgjdbc

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

If you have an SQLException *and* it can be cast to [org.postgresql.util.PSQLException][psqle], then you can call getServerErrorMessage() on it, and get a [ServerErrorMessage][sem]. Same deal if you have an

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Just as in PL/-Java, the design here with a single PSQLException class is an obstacle to moving forward with the categorized

Update: This turns out to be more timely than I realized— PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[psqle]: https://jdbc.postgresql.org/development/privateapi/index.html?org/post
[psqlw]: https://jdbc.postgresql.org/development/privateapi/index.html?org/post
[sem]: https://jdbc.postgresql.org/documentation/publicapi/index.html?org/po
[pgjdbc-docs]: https://jdbc.postgresql.org/documentation/head/index.html
#####
pgjdbc-ng

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

In pgjdbc—ng, categorized exceptions are partially implemented: at least there is one [PGSQLIntegrityConstraintViolationException][pgsicve], and one [PGSQLSimpleException][pgsse] for everything else.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Interestingly, the interface gives access *only* to the column, constraint, datatype, schema, and table names available from PostgreSQL 9.3 on-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

All of the elements the protocol can forward are available on a different object, [com.impossibl.postgres.protocol.Notice][notice], but I am not sure user code

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Unlike both PL/-Java's [ErrorData][edata] and pgjdbc's [ServerErrorMessage][sem], in pgjdbc—ng both the [PGSQLExceptionInfo][pgsei] and the [Notice][notice] are mutable, providing setter methods as

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[pgsicve]: http://impossibl.github.io/pgjdbc-ng/apidocs/0.6/index.html?com/impossibl/postgres/jdbc/PGSQLIntegri
[pgsse]: http://impossibl.github.io/pgjdbc-ng/apidocs/0.6/index.html?com/impossibl/postgres/jdbc/PGSQLSimple
[pgsei]: http://impossibl.github.io/pgjdbc-ng/apidocs/0.6/index.html?com/impossibl/postgres/api/jdbc/PGSQLEx
[notice]: http://impossibl.github.io/pgjdbc-ng/apidocs/0.6/index.html?com/impossibl/postgres/protocol/Notice.htm

---

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

###
Orig-
inat-
ing
log-
gable
events

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Whether running client-side or in a server-side PL, it's often helpful to look at the details of events from below, as the

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Here again, PL/pgSQL makes a good example. Using [RAISE][], code can generate an event with direct control of eleven of its

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[RAISE]: http://www.postgresql.org/docs/current/static/plpgsql-errors-and-messages.html

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PL/pgSQL has hit a kind of sweet spot with syntax that is so easy and clear it invites writing good messages that an ultimate user could

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

RAISE invalid_text_representation USING MESSAGE = 'Unrecognized prefix in telephone number' || text, DETAIL = 'The digits at

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

is about as clear as can be in the code, as well as giving anyone who receives it a fighting chance at understand-

---

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

####
As
for
PL/-
Java

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PL/Java is not yet at parity with PL/pgSQL on this dimension. If PL/-Java code *catches* any exception that began as a Post-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

For any other kind of exception, only message is set (from the exception class name and message), and SQLState of XX000 for "in-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The other way for PL/-Java code to originate a log event is to use the logging API. PL/-Java presents mostly Java standard APIs

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

When passed on to PostgreSQL, the details include the timestamp, class name or logger name, the message, and the stack trace of

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[jlog]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/summary.html
[jlvl]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/
[scn]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/
[smn]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/

#####

Mapping of severity levels

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Because there are only seven predefined [java.util.logging.Level][jlvl]s and some of their names are different from PostgreSQL's, PL/Java maps them as follows:

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

| |
|FINEST|FINER|FINE|
|
|INFO|
|WARN-
ING|SEVERE|
| |
|—
|—
|—
|—
|—
|—
|—
|—
|—
|—
|—
|—
|—|
|DE-
BUG5|DEBUG4|DEBUG3|DEBUG2|DEBUG1|LOG|COMMERROR|IN

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The Java level CONFIG isn't explicitly mapped, and anything that isn't explicit will map to the PostgreSQL level LOG. For completeness,

---

Update:

This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

####

How can JDBC front-end code originate events?

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

While the front-end situation may be simpler (there is no need to make logging interoperate with server code in both di-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

These days, there could even be another sophisticated layer or three sitting on top of JDBC and beneath the applica-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

A day that I would like to see—and I think it can be reached—is the day when a PostgreSQL error can be raised by

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

This becomes even more appealing, and maybe even more achievable, if PL/-Java and a front-end JDBC work toward sharing more

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.
#####
when
throw-
ing

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Being JDBC interfaces, both pgjdbc and pgjdbc-ng throw the standard JDBC SQLException (or, more precisely, subclasses of it), and create instances

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

JDBC categorized exceptions are not yet supported by pgjdbc, and are partly supported by pgjdbc—ng.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The pgjdbc—ng exceptions that implement [PGSQLExceptionInfo][pgsei] can be instantiated from scratch, and can have the column, constraint, datatype, schema,

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The pgjdbc [PSQLException][psqle] and [PSQLWarning][psqlw] throwables can be instantiated from scratch, and can be constructed from a [ServerErrorMessage][sem] that can also be built

---

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

#####
when
log-
ging

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Logging in **pgjdbc**, which is older than java.util.logging, is done with the project-specific org.postgresql.core.Logger. This simple class identifies messages with a connection

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

When logging a PSQLException instance that carries a ServerErrorMessage, the result looks much like a message logged by the backend, because

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[gt]: https://jdbc.postgresql.org/development/privateapi/index.html?org/post
[rb]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/Resourc

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Logging **in pgjdbc—ng** is done using java.util.logging, as in PL/Java, and in keeping with the appearance of the java.util.logging-related API in JDBC itself

---

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[gpl]: http://docs.oracle.com/javase/7/docs/api/index.html?java/sql/Driver.ht
[nex]: http://impossibl.github.io/pgjdbc-ng/apidocs/0.6/index.html?com/impossibl/postgres/system/NoticeExcep
[erut]: http://impossibl.github.io/pgjdbc-ng/apidocs/0.6/index.html?com/impossibl/postgres/jdbc/ErrorUtils.htm

What would a nice API look like?

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

By this point it should be clear why I've been writing about "log events" as one concept, when I112 might be expected to talk of

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

This shapeshifting is not only possible but downright common in PostgreSQL and JDBC, and especially in PL/-Java, where the same event

---

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

An abstract LogRecord class (not derived from Exception)

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

We've seen that all three of (pgjdbc, pgjdbc–ng, PL/Java) include a class of some sort ([ServerErrorMessage][sem], [Notice][notice], and [ErrorData][edata], respectively) that is meant to carry

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

So, my proposal starts here: there should be such a class, and it should be documented and available as PostgreSQL extended JDBC

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

###
An
in-
ter-
face
for
ex-
cep-
tions
that
carry
LogRecords

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

To allow moving forward with the [categorized exceptions][catex] in JDBC 4.0, there needs to be an interface rather than

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

Different concrete subclasses of LogRecord

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PL/Java would supply its special concrete implementation that wraps a native error-data block; a front-end JDBC would supply

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

An ereport-like API for creating a LogRecord from scratch

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

…
start-
ing
with
a
static
method
on
LogRecord
and
with
method
chain-
ing:
import static org.postgresql.something.LogRecord.ereport; ... logrec =

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

Methods for the conversions into/out of log system or exception

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Examples log() and throwAs() were seen above, with throwAs a convenience built on asException(...). If the LogRecord has been freshly constructed, asException creates the

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The static fromException() method does the reverse: if the exception was created from a LogRecord originally (so, it implements the

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Those methods are what make possible the repeated batting around that an event might live through on its way from a deep call stack

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

In concert with existing standard API

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

I
pro-
pose
to
con-
verge
on
[java. util .logging][jlog],
and
for
this
ex-
tended
LogRecord
class
to
be
de-
rived
from
the
stan-
dard
[LogRecord][logrec].

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

(Soon
be-
low
I
will
touch
on
how
to
“con-
verge
on
java. util .logging”
with-
out
seri-
ous
dis-
rup-
tion
of
pgjdbc,
which
cur-
rently
uses
the
home-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The specialized class will have several extra methods, and some overridden ones just to give it a reasonable default

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

None
of
those
dif-
fer-
ences
stop
it
from
be-
ing
a
valid
in-
stance
of
java. util .logging.LogRecord,
and
it
can
be
passed
into
the
log-
ging
sys-
tem

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

Call
sites
that
aren't
try-
ing
to
make
good
user-
visible
mes-
sages
(all
the
usual
logger.finest("sent an M, got two dollar signs and a comma")
kind
of
thing)
don't
have
any
need
to
change.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[gmsg]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/
[logrec]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/
[lgr]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/
[logr]: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/logging/

Using familiar level names

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The current implementation in PL/-Java maps PostgreSQL severity levels onto the (smaller set of) standard [Level][jlvl]s. This adds

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
SET log_min_messages TO DEBUG2; SELECT javatest.logmessage('F
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

are talking about the same severity level, and it's an error to forget and use the other name either place, and the map-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

A feature of the design of [Level][jlvl] is it can be subclassed, and additional levels can be defined; the numeric values

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PostgreSQL	Java
ALL	
FINEST?	DEBUG5
FINEST?	DEBUG4
DEBUG3	
FINER	DEBUG2
FINE	DEBUG1
CONFIG	

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

As you can see, I'm still considering arguments about where FINEST and SEVERE should go.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The effect, again, is that code from elsewhere that only expects the usual names from the standard library will work fine, code with

---

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

###
Could
pgjdbc
move
to
java. util .logging
with-
out
dis-
rup-
tion?

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

I think so. The class org.postgresql.core.Logger could be kept, and simply delegate to other classes; the changes at points where log events are read off the

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

I think the categorized exceptions in JDBC 4 are worth moving to, and offer a much nicer way for client code to distin-

---

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

##
A
place
to
pause

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

I have not managed to squeeze in every relevant thought here, but this is already long and enough to elicit some discussion and

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Tips for resolving build problems

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Some typical issues encountered when building PL/-Java can be listed here, along with tips for resolving them.

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

##
The
tips
that
al-
ways
ap-
ply

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Please do carefully read the build instructions, especially the "software prerequisites" section, and the "special topics" section,

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

bld:
https://tada.github.io/pljava/build/build.html

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Also be sure to review the "troubleshooting the build" section at the end of the build instructions page.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

If you review [the mailing list archive][pljdva] and the [issues list][issues], you may find a report of a situation like your own. (On the

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

[pljdva]:
http://lists.pgfoundry.org/pipermail/pljava-
dev/
[is-
sues]:
https://github.com/tada/pljava/issues
##
Fail-
ure
shown
for
pljava—so

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

Missing –devel prerequisite packages

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The most common cause of reported failures building pljava—so is a missing required file. Sometimes your distribution's packaging

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The solution is simple: look over the error messages from the pljava—so section of the build output to find any that refer

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Find out the name of the package, according to the OS or package distribution you are using, that contains

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Further tip: Finding the error message that really mattered is easier if you follow the "troubleshooting the build" tip about

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

##
Still
stuck?
Please
de-
scribe
the
issue
you
are
fac-
ing
on
the
mail-
ing
list.

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

pljdv:
http://lists.pgfoundry.org/mailman/listinfo/pljava-
dev
#
Trig-
gers

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The method signature of a trigger is predefined. A trigger method must always return void and have a org.postgresql.pljava.TriggerData parameter.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The Result-Sets are only available for triggers that are fired ON EACH ROW. Delete triggers have no new row, and insert trig-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

In addition to the sets, several boolean methods exists to gain more information about the trigger.

```sql
CREATE TA-
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
CREATE
FUNC-
TION
mod-
date-
time()
RE-
TURNS
trig-
ger
AS
'org.postgresql.pljava.example.Triggers.moddatetime'
LAN-
GUAGE
java;
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
CREATE TRIGGER mdt_moddatetime
BEFORE UPDATE ON mdt
FOR EACH ROW EXECUTE PROCEDURE moddatetime (moddate);
```

And here is

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```java
/** *
Update a modification time when the row is updated.
*/
static void modtime(TriggerData td) throws SQLException {
if(td.isFiredForStatement())
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
if(td.isFiredAfter())
throw
new
Trig-
gerEx-
cep-
tion(td,
“must
be
fired
be-
fore
event”);
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
if(!td.isFiredByUpdate())
throw
new
Trig-
gerEx-
cep-
tion(td,
“can
only
pro-
cess
UP-
DATE
events”);
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
ResultSet _new = td.getNew();
String[] args = td.getArguments();
if(args.length != 1)
throw new TriggerException(td,
"one argument was expected");
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

_new.updateTimestamp(args[0],
new
Times-
tamp(System.currentTimeMillis()));
} “‘
#
User
guide
(wiki
ver-
sion)

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The first reference should be the [user guide at the main project site][ug].

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Here at this wiki version, you may still find useful information that is not yet migrated to the project site. Some of

---

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

[ug]:
https://tada.github.io/pljava/use/use.html
##
Util-
ities

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* The [PL/-Java Deployer][dplr] is a Java client program that helps you deploy PL/-Java in the database. *It is now obsolescent; for cur-*

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

igd:
https://tada.github.io/pljava/install/install.html
[dplr]:
https://tada.github.io/pljava/pljava-
deploy/apidocs/index.html?org/postgresql/pljava/deploy/Deployer.html
##
Au-
thor-
ing

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* [[Writing Java functions, triggers, and types]]
* Using a [[SQL deployment descriptor]]
* [[Security]]

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Debugging

* [[Debugging your Java code]]
* [[Debugging in C]]

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.
##
Trou-
bleshoot-
ing
*
[[Spo-
radic
hang-
ing]]
#
Us-
ing
JDBC

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PL/Java contains a JDBC driver that maps to the PostgreSQL SPI functions. A connection that maps to the current transaction

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* The transaction cannot be managed in any way. Thus, you cannot use methods on the connection such as: *

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

*

[[Function mapping]]

*

[[Triggers]]

*

[[Default Type Mapping]]

*

[Mapping an SQL type to a Java class]

*

[[Creating a Scalar

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

In order to completely uninstall PL/-Java you need to have superuser privileges on the database. Here's how you do it.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

0. Get rid of the sqlj schema and all objects depending on it.

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* If you installed PL/Java with CREATE EXTENSION pljava then drop it with DROP EXTENSION pljava CASCADE
* If you installed PL/Java with a184 LOAD command, then drop it with

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Caution: Either command will drop the PL/-Java schema and language declarations, all jars you may have loaded, all functions and types

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

You can try either command *without* CASCADE first, to see a list of what would be dropped.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

0. Remove any settings of PL/-Java variables (configuration variables with names starting with pljava.) that you may have changed

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* If you had set a variable *var* for a particular database using ALTER DATABASE dbname SET var ... then reset it using ALTER DATABASE dbname RESET var.

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* If you had set it for the whole cluster using ALTER SYSTEM SET var ... then reset it using ALTER SYSTEM RESET var and, when you have reset all, use SELECT pg_reload_conf().

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* If you had set PL/-Java variables by editing the configuration file (particularly on PostgreSQL before 9.2, where this

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* The variable dynamic_library_path is not specific to PL/-Java, but if you added a directory to it for the sake of PL/-Java, undo that.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

0. Remove the PL/-Java files from the file system.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

* If you installed PL/-Java with a package manager, uninstall it the same way.
* Otherwise, remove the installed files

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PL/Java is an open source project and contributions are vital for its success. In fact, all development of the project is done

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Getting started

* Make sure you have a GitHub account.
* Create a fork of the PL/Java repository.
* Take a

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Let people know what you're planning

You should let the community know what you're planning to do by discussing it on the

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Making Changes

* Create a local clone of your fork.
* Create a topic branch for your work. You should branch off the *master*

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Submitting Changes

* Push your changes to a topic branch in your fork of the repository.
*198 Submit a pull request to

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Commit Message Format

What should be included in a commit message? The three basic things to include are:

* Sum-

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

Here
is a
sam-
ple
com-
mit
mes-
sage
with
all
that
in-
for-
ma-
tion:

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Some POM's did not have the source encoding specified. This caused unnecessary warning printouts during build. This commit

---

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

Closes
#1234

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The issue number is optional and should only be included when the commit really closes an issue. The close will then oc-

---

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Creating a scalar (or, base) user-defined type

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

This text assumes that you have some familiarity with how scalar types are created and added to the PostgreSQL type system.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

[xtypes]: http://www.postgresql.org/docs/8.4/static/xtypes.html

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Creating new scalar type using Java functions is very similar to how they are created using C functions from an SQL per-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Let us create a type called javatest.complex (similar to the complex type used in the PostgreSQL docu-

mentation). The name of

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

##
The
Java
code
for
the
scalar
type

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

Prerequisites for the Java implementation

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The java class for a scalar UDT must implement the java.sql.SQLData interface. In addition, it must also implement a method static T parse(String stringRepresentation, String typeName) where

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

“‘java
pack-
age
org.postgresql.pljava.example;

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
import
java.io.IOException;
im-
port
java.io.StreamTokenizer;
im-
port
java.io.StringReader;
im-
port
java.sql.SQLData;
im-
port
java.sql.SQLException;
im-
port
java.sql.SQLInput;
im-
port
java.sql.SQLOutput;
im-
port
java.util.logging.Logger;
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
import
org.postgresql.pljava.annotation.Function;
im-
port
org.postgresql.pljava.annotation.SQLType;
im-
port
org.postgresql.pljava.annotation.BaseUDT;
import
static
org.postgresql.pljava.annotation.Function.Effects.IMMUTABLE;
im-
port
static
org.postgresql.pljava.annotation.Function.OnNullInput.RETURNS_NUI
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
@BaseUDT(schema="javatest",
name="complex",
internalLength=16,
alignment=BaseUDT.Alignment.DOUBLE)
public class ComplexScalar implements SQLData
{ private double m_x;
private double
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
@Function(effects=IMMUTABLE,
on-
NullInput=RETURNS_NULL)
pub-
lic
static
Com-
plexS-
calar
parse(String
in-
put,
String
type-
Name)
throws
SQLEx-
cep-
tion
{ try
{216
Stream-
Tok-
enizer
tz =
new
Stream-
Tok-
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
public ComplexScalar() { }
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
public ComplexScalar(double x, double y, String typeName)
{
m_x = x;
m_y = y;
m_typeName = typeName;
}218
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
@Override
pub-
lic
String
get-
SQL-
Type-
Name()
{ re-
turn
m_typeName;
}
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
@Function(effects=IMMUTABLE,
on-
NullInput=RETURNS_NULL)
@Over-
ride
pub-
lic
void
read-
SQL(SQLInput
stream,
String
type-
Name)
throws
SQLEx-
cep-
tion
{
m_x
=220
stream.readDouble();
m_y
=
stream.readDouble();
m_typeName
=
type-
```

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
@Function(effects=IMMUTABLE,
on-
NullInput=RETURNS_NULL)
@Over-
ride
pub-
lic
void
writeSQL(SQLOutput
stream)
throws
SQLEx-
cep-
tion
{
stream.writeDouble(m_x);
stream.writeDouble(m_y);
}
```

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
@Function(effects=IMMUTABLE,
onNullInput=RETURNS_NULL)
@Override
public String toString()
{
s_logger.info(m_typeName + ”toString”);
StringBuffer sb = new StringBuffer();
sb.append(‘(’);
sb.append(m_x);
sb.append(‘,’);
sb.append(m_y);
sb.append(‘)’);
return sb.toString();
}
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

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

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The class itself is annotated with @BaseUDT, giving its SQL schema and name, and the length and alignment needed for its in-

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Because the compiler knows the class is a BaseUDT, it already expects the parse, toString, readSQL, and writeSQL methods to be present, and will gener-

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

#
De-
bug-
ging
PL/-
Java
C
code

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Ensure the native code is compiled for debugging

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Debugging is much more pleasant when the C code has been compiled with debugging information included. Edit the pljava—so/pom.xml file,

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

xml <configuration> ... <c> ... <debug>true</debug> <defines> ... <

Save the pom.xml file and rebuild PL/-Java (or just the pljava—so subproject, to save time).

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Start PL/-Java and attach a debugger

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Start psql and set the PL/-Java debug flag, and issue a call to some Java function.

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
sql set pljava.debug to on; select sqlj.get_classpath();
```

You will see a message resembling this:

```
INFO: Backend pid = 2830. Attach the debugger and set pljavaDebug to
```

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Use another window and attatch gdb or another debugger.

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

```
sh gdb <full path to the postgres executable> <your Backend pid>
```

The debugger will break into the PL/-Java code while it is in a dummy loop. You can break this loop by setting the global vari-

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

gdb (gdb) set pljavaDebug=0 (gdb) <set breakpoints etc. here> (gdb) c
That's it!

Debugging with dbx

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Copied from [Debugging PL/-Java Applications with Solaris Studio dbx][dpjdbx], Johann 'Myrkraverk's blog on my.opera.

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

[dpjdbx]:
http://my.opera.com/myrkraverk/blog/2010/12/11/debugging-
pl-
java-
with-
dbx
###
Set-
ting
up
the
Server

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Debugging PL/-Java code requires debugging of the server process itself. This means the debugger must be run as the same

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

As per the dbx manual, the Java virtual machine must be started with the options −Xdebug −Xnoagent −Xrundbx_agent. This can be done by having the

Update:
This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

This means the jvm will load libdbx_agent.so whose location must be in the server's runtime load path (LD_LIBRARY_PATH).

Solaris Studio 12.2 manual

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

properties LD_LIBRARY_PATH_64=/opt/solstudio12.2/lib/dbx/amd in the server's environment2 where Studio is installed in /opt.

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

###
Set-
ting
up
the
de-
bug-
ger
(dbx)

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

PL/Java loads classes from the database which dbx does not know about so it must be told where the jar files can be found. This is done with the CLASS

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

properties CLASSPATHX=/home/johann/src/Java/PLJava/Hello.jar

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

which must be set in the debugger's environment. In addition it must also be told where to find the Java source files. For

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

To
de-
bug
PL/-
Java
it-
self
we
need
its
source
path
in
JAVAS-
RC-
PATH
too.

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

###

Attaching dbx to the Server's Process

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

Before we attach to the server we need to make sure that PL/-Java has been loaded and that the virtual machine has been cre-

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

properties (dbx) stop in com.myrkraverk.Hello.hello dbx: ”com” is not c

Update: This turns out to be more timely than I realized—PL/Pythonu is also [developing a patch][plpu] (starting last month) to the same end.

The best way is to run some simple Java function before we attach the debugger. I250 psql session one way is to run

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

```
sql CREATE FUNCTION getsysprop(VARCHAR) RETURNS VARCH
```

Now
it is
just
a
mat-
ter
of
get-
ting
the
server's
pid

Update:
This
turns
out
to
be
more
timely
than
I
realized—
PL/Pythonu
is
also
[de-
vel-
op-
ing a
patch][plpu]
(start-
ing
last
month)
to
the
same
end.

“‘psql
jo-
hann=#
se-
lect
pg_backend_pid();
pg_backend_pid

10767

(1 row)

and attach dbx.

```sh
$ dbx - 10767
Reading postgres
Reading ld.so.1
```

```
Reading libxslt.so.1
Output elided.
Reading libjava.so
Reading libzip.so
Attached to process 10767 with 10 LWPs
t@1 (l@1) stopped in __so_recv at 0xfffffd7fff23d14a
0xfffffd7fff23d14a: __so_recv+0x000a:   jae      __so_recv+0x16 [ 0xfffffd7fff23d
Current function is secure_read
  303           n = recv(port->sock, ptr, len, 0);
(dbx)
```

Debugging our Java code

Our "hello world" is very simple.

```
package com.myrkraverk;

class Hello
{
    public static int hello()
    {
   return 17;
    }
}
```

Assuming we have already compiled (with -g) and jar archived our code3 we can tell dbx to stop in our method whether we have run sqlj.install_jar() first or not.

```
(dbx) stop in com.myrkraverk.Hello.hello
(2) java stop in com.myrkraverk.Hello.hello()
```

And if not, we just detach dbx, re-compile/re-archive and place it where dbx can find it before we attach again.

And of course we have to let the server continue running.

```
(dbx) cont
```

In our psql session, we can now4 load our class into the database,

```
johann=# select sqlj.install_jar('file:///home/johann/src/Java/PLJava/Hello.jar
 install_jar
-------------


(1 row)
```

set the classpath

```
johann=# select sqlj.set_classpath( 'johann', 'Hello' );
 set_classpath
---------------


(1 row)
```

and create the sql function.

```
johann=# create function hello() returns int4
  as 'com.myrkraverk.Hello.hello' language java;
CREATE FUNCTION
```

Now when we run it,

```
johann=# select hello();
```

dbx halts at the breakpoint.

```
stopped in com.myrkraverk.Hello.hello at line 14 in file "Hello.java"
   14       return 17;
```

Final Notes

It is outside the scope of this tutorial to teach debugging Java applications with dbx. See the Solaris Studio manual for the details.

Download

Download the hello world source code from my.opera. Boost Licensed.

Footnotes

- 1 It's been fashionable lately to use the pronoun "her" in these cases. The author firmly believes the pronoun's gender should be chosen as the writer's gender however.
- 2 This means the environment the postgres command is run in.
- 3 And that dbx can find it, as described above.
- 4 Or before, it doesn't matter.

Debugging with jdb

PL/Java is debugged like any other Java application using JPDA. Here is an example of how to set it up using the PostgreSQL psql utility and the bundled command line debugger jdb (you will probably use your favourite IDE instead but the setup will be similar).

Let's assume we want to debug the SQL function javatest.testSavepointSanity() and that the function is mapped to the java method org.postgresql.pljava.example.SPIActions.testSavepointSa (the example is from the examples.jar found in the PL/Java source distribution).

Fire up psql and issue the following commands:

```
SET pljava.vmoptions TO '-agentlib:jdwp=transport=dt_socket,server=y,address=844
SELECT javatest.testSavepointSanity();
```

Now your application hangs. In the server log you should find a message similar to:

```
Listening for transport dt_socket at address: 8444
```

Use another command window and attach your remote debugger:

```
jdb -connect com.sun.jdi.SocketAttach:port=8444 \
    -sourcepath /home/workspaces/org.postgresql.pljava/src/java/examples
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
&gt;
VM Started: No frames on the current call stack

main[1]
```

This means that the debugger has attached. Now you can set breakpoints etc.:

```
main[1] stop in org.postgresql.pljava.example.SPIActions.testSavepointSanity
Deferring breakpoint org.postgresql.pljava.example.SPIActions.testSavepointSanit
It will be set after the class is loaded.
main[1] cont
> Set deferred breakpoint org.postgresql.pljava.example.SPIActions.testSavepoint

Breakpoint hit: "thread=main", org.postgresql.pljava.example.SPIActions.testSave
78              Connection conn = DriverManager.getConnection("jdbc:default:conn

main[1]
```

Now it's up to you…

Scalar types

Scalar types are mapped in a straight forward way. Here's a table of the current mappings (will be updated as more mappings are implemented).

PostgreSQL

Java

bool

boolean

"char"

byte

int2

short

int4

int

int8

long

float4

float

float8

double

char

java.lang.String

varchar

java.lang.String

text

java.lang.String

name

java.lang.String

bytea

byte[]

date

java.sql.Date

time

java.sql.Time (stored value treated as local time)

timetz

java.sql.Time

timestamp

java.sql.Timestamp (stored value treated as local time)

timestamptz

java.sql.Timestamp

Array scalar types

All scalar types can be represented as an array. Although PostgreSQL will allow that you declare multi dimensional arrays with fixed sizes, PL/Java will still treat all arrays as having one dimension (with the exception of the bytea[] which maps to byte[][]). The reason for this is that the information about dimensions and sizes is not stored anywhere and not enforced in any way. You can read more about this in the PostgreSQL Documentation.

However, the current implementation does not enforce the array size limits — the behavior is the same as for arrays of unspecified length.

Actually, the current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring number of dimensions or sizes in CREATE TABLE is simply documentation, it does not affect run-time behavior.

PostgreSQL

Java

bool[]

boolean[]

“char”[]

byte[]

int2[]

short[]

int4[]

int[]

int8[]

long []

float4[]

float[]

float8[]

double[]

char[]

java.lang.String[]

varchar[]

java.lang.String[]

text[]

java.lang.String[]

name[]

java.lang.String[]

bytea[]

byte[][]

date[]

java.sql.Date[]

time[]

java.sql.Time[] (stored value treated as local time)

timetz[]

java.sql.Time[]

timestamp[]

java.sql.Timestamp[] (stored value treated as local time)

timestamptz[]

java.sql.Timestamp[]

Domain types

A domain type will be mapped in accordance with the type that it extends unless you have installed a specific mapping to override that behavior.

Pseudo types

PostgreSQL

Java

“any”

java.lang.Object

anyelement

java.lang.Object

anyarray

java.lang.Object[]

cstring

java.lang.String

record

java.sql.ResultSet

trigger

org.postgresql.pljava.TriggerData (see [[Triggers]])

NULL handling of primitives

The scalar types that map to Java primitives can not be passed as null values. To enable this, those types can have an alternative mapping. You enable this mapping by explicitly denoting it in the method reference.

```
CREATE FUNCTION trueIfEvenOrNull(integer)
  RETURNS bool
  AS 'foo.fee.Fum.trueIfEvenOrNull(java.lang.Integer)'
  LANGUAGE java;
```

In java, you would have something like:

```
package foo.fee;

public class Fum
{
  static boolean trueIfEvenOrNull(Integer value)
  {
    return (value == null)
      ? true
      : (value.intValue() % 1) == 0;
  }
}
```

The following two statements should both yield true:

```
SELECT trueIfEvenOrNull(NULL);
SELECT trueIfEvenOrNull(4);
```

In order to return null values from a Java method, you simply use the object type that corresponds to the primitive (i.e. you return java.lang.Integer instead of int). The PL/Java resolver mechanism will find the method regardless. Since Java cannot have different return types for methods with the same name, this does not introduce any ambiguities.

Starting with PostgreSQL version 8.2 it will be possible to have NULL values in arrays. PL/Java will handle that the same way as with normal primitives, i.e. you can declare methods that use a java.lang.Integer[] parameter instead of an int[] parameter.

Composite types

A composite type will be passed as a read-only java.sql.ResultSet with exactly one row by default. The ResultSet will be positioned on its row so no call to next() should be made. The values of the composite type are retrieved using the standard getter methods of the ResultSet. Example:

```
CREATE TYPE compositeTest
  AS(base integer, incbase integer, ctime timestamptz);

CREATE FUNCTION useCompositeTest(compositeTest)
  RETURNS VARCHAR
  AS 'foo.fee.Fum.useCompositeTest'
  IMMUTABLE LANGUAGE java;
```

In class Fum we add the static following static method The foo.fee.Fum.useCompositeTest method:

```
public static String useCompositeTest(ResultSet compositeTest)
throws SQLException
{
  int base = compositeTest.getInt(1);
  int incbase = compositeTest.getInt(2);
  Timestamp ctime = compositeTest.getTimestamp(3);
  return "Base = \"" + base +
    "\", incbase = \"" + incbase +
    "\", ctime = \"" + ctime + "\"";
}
```

Default mapping

Types that have no mapping are currently mapped to java.lang.String. The standard PostgreSQL textin/textout routines registered for respective type will be used when the values are converted.

Exception handling

You can catch and handle an exception in the PostgreSQL back-end just like any other exception. The back-end ErrorData structure is exposed as a property in a ServerException class derived from java.sql.SQLException, and the Java try/catch mechanism is synchronized with the back-end mechanism.

Note: for several reasons (see [Thoughts on logging] for background), referring to ServerException and ErrorData from your code is not currently recommended, and in the future may become impossible. An improved mechanism is expected in a future release. Until then, using only the standard Java API

of java.sql.SQLException and its standard attributes (such as SQLState) is recommended wherever possible.

PL/Java will always catch exceptions that you don't. They will cause a PostgreSQL error and the message is logged using the PostgreSQL logging utilities. The stack trace of the exception will also be printed if the PostgreSQL configuration parameter log_min_messages is set to DEBUG1 or lower.

Important Note:

You will not be able to continue executing back-end functions until your function has returned and the error has been propagated when the back-end has generated an exception unless you have used a save-point. When a save-point is rolled back, the exceptional condition is reset and execution can continue.