

Large Customers Want PostgreSQL Too!

Ken Rosensteel

LIBERATE IT

Who is Bull Information Systems

- 8000 Employees Worldwide
- \$1.6 Billion Annual Revenue
- Headquartered in Paris
- Servers / Services / Supercomputers / Open Source
 - R&D Partners with Intel, IBM, EMC, Oracle, Microsoft,.....
- Phoenix, Arizona R&D
 - Mainframe / Open System Architects
 - Intel / IBM chips, EMC Storage, Cisco networks,
 - Database Integration/Migration Services
 - Oracle, IMS/IDS2, DB2, Postgres, SQL Server,.....
 - Targeting Fortune 500 customers

Large Customers Discussed in the Presentation

CNAF

- **Migrating from Mainframe Relational Database**

CAIXA Bank and Banco do Brasil

- **Migrating from Oracle to PostgreSQL**

CNAF is a Very Large Organization

- Social Security for France
- 35,000 employees
 - ✓ 40 million phone calls / year
 - ✓ 20 million person contacts / year
 - ✓ 170 million letters (sent & received)
 - ✓ 30 million beneficiaries
- 70 billion € of Benefits paid Annually

Challenge

- Convert the OLTP/Batch Applications of CNAF from Mainframe COBOL to use Postgres

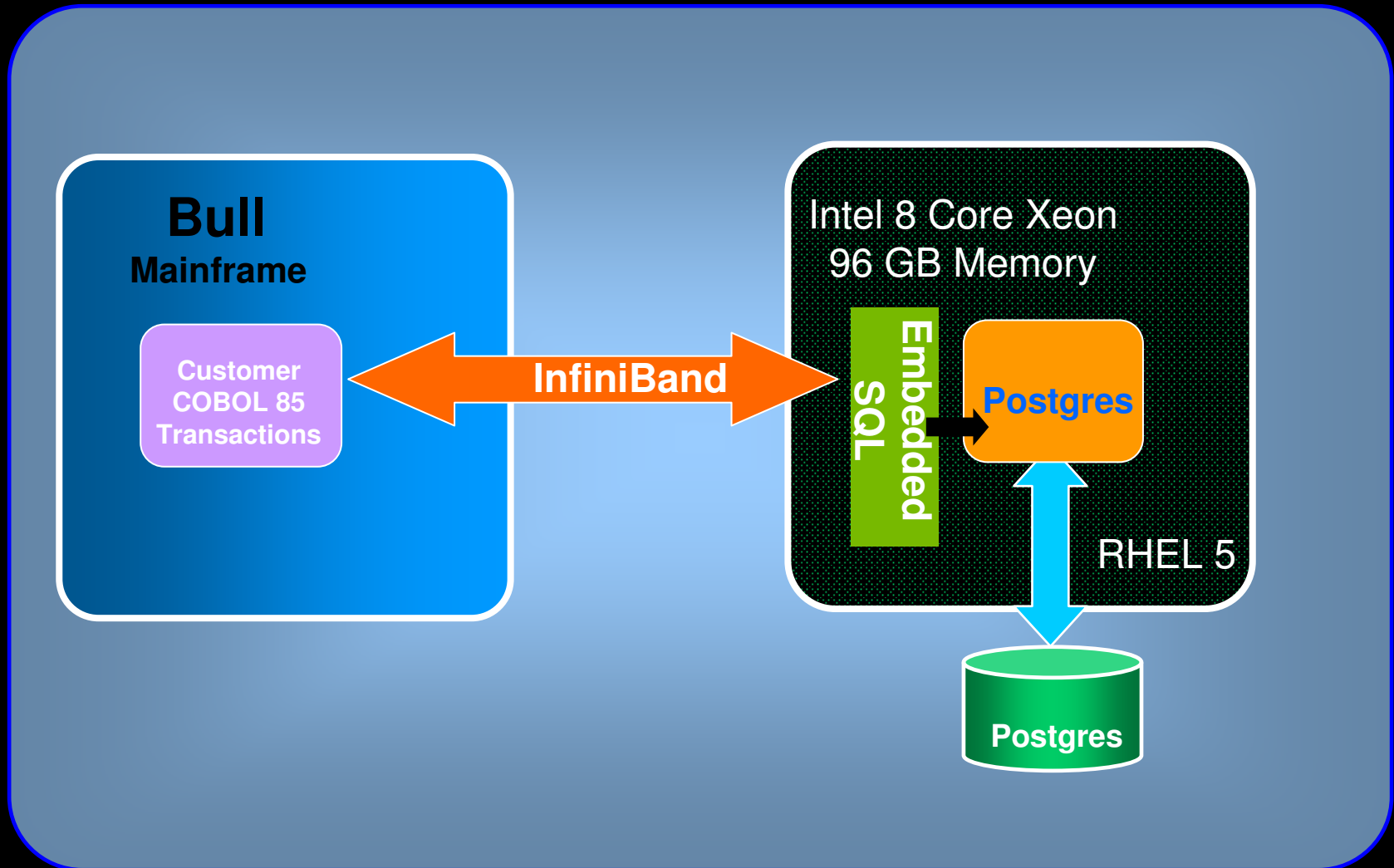
Issue #1 – It's Cobol / It's Mainframe

- Issue #1 It's Cobol – 1150 Modules
 - Coded to DB2 Standard for Embedding SQL in Cobol
- Desire to keep COBOL on Mainframe
 - Just change the database access from DB2 to Postgres
 - Minimize Cost and Impact to Programs
- No Postgres Port to Bull Mainframe

Solution:

- Postgres Cobol Preprocessor
 - Cobol Logic remains on Mainframe
 - Embedded SQL Modules generated for Linux (C language)
- Client / Server from Mainframe Cobol to Postgres on Xeon
 - InfiniBand Connection (low latency) from Mainframe to Xeon
 - Postgres on RedHat on Intel Server

Mainframe Cobol to Postgres on Linux



Issue #2 - Cobol Application Performance

- Tens of Thousands of OLTP Users
- .2 seconds average response time
- Batch programs confined to 12 hour Nightly “Window”
- Customer Batch Benchmarks Provided

Postgres Performance Improvements

Impact to Customer Batch Application

Elapsed Time
(seconds)

Original Application Target	415
Initial Postgres Measurement	751
(after SQL Tuning, I/O tuning,...all the normal stuff)	
Enhanced Postgres	381

Analyzing an application you didn't write

Application comprised of 500 Cobol Modules with thousands of SQL included

Module	STM T #	TYPE	#EXECS	TOTAL TIME (s)	TIME/STMT (s)
CDAVP0	21	OPEN	1849346	1158	0.000626
CDZZ0	5	OTHER	1439169	877	0.000609
CD0006	3	OPEN	518507	854	0.001646
CDPER0	22	OPEN	553611	751	0.001357
CDPET0	23	OPEN	1089061	695	0.000637
CDPE70	18	OPEN	788098	615	0.000779
CDDRC0	32	OPEN	863386	572	0.000662
CDBUFC	11	OPEN	52774	569	0.010783
CDBUFC	14	OPEN	165996	514	0.003095
CDDRC0	31	OPEN	587565	494	0.000840

Caching Prepared Statements

- For batch, many of the SQL statements were performed thousands of times.
- Elapsed time for repeated operations in Batch indicated lack of SQL Cache for the query plan
 - SQL Cache exists in server, but not used by the Embedded SQL interface
 - With help from Simon Riggs and Michael Meskes, we implemented “-r prepare” option for ECPG
 - Prepared Queries are cached
 - Performance impact - 13 % Reduction to application elapsed time

Minimizing Server Interactions

- **Cursor Operations showed high overhead for FETCH operations**
 - **Overhead of Server Interaction for each Fetch**
 - **Used Cobol Preprocessor to Replace Cursor operation with SELECT INTO and memory array**
 - **Client library (libpq) on SELECT gets many rows from Server on one call**
 - **Passes pointer to application versus data copy**
 - **Impact 27% elapsed time of Fetch intensive test and 9% to customer batch benchmark**

cursor is translated to the following SELECT INTO query:

```
SELECT lastName  
  INTO :ptrhv_lastName  
  FROM participant
```

the host variable defined as:

```
char lastName[30];
```

redefined as:

```
static char **ptrhv_lastName;
```

FETCH is emulated by iterating through the pointers to access the result set:

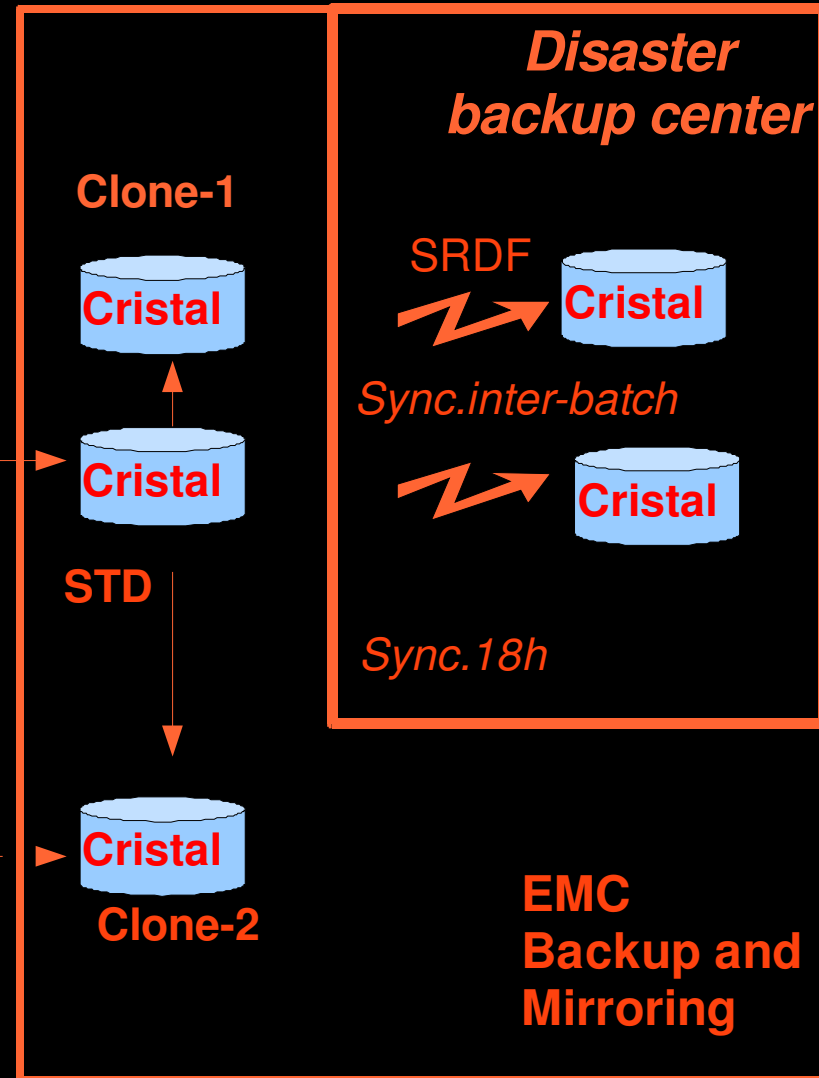
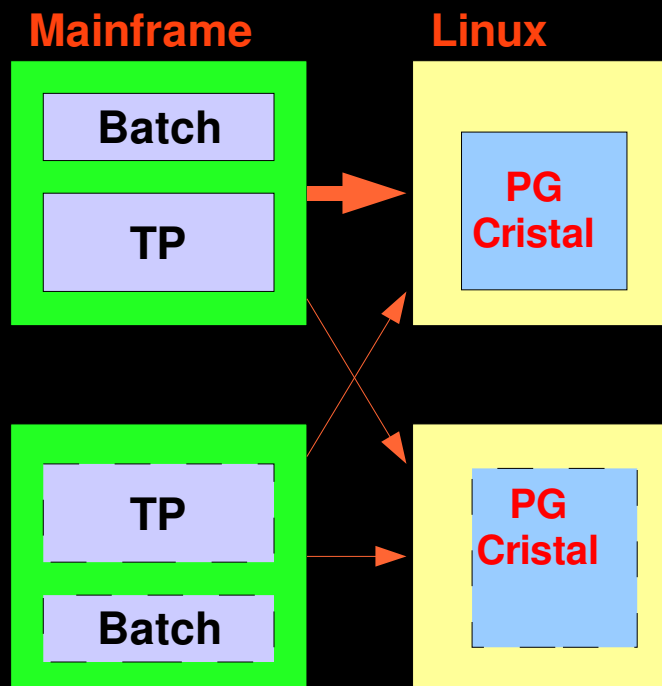
```
strncpy(lastName, ptrhv_lastName[ix], dLength);
```

Issue #3 System Operation

- Mindset of Customer is Mainframe
 - EMC Storage (>\$500K Storage Cabinet)
 - EMC Timefinder Block Level backups
 - Disaster Recovery Site using SRDF
 - High Availability using Mirroring using Backup Storage Node
- Challenges
 - Minimal storage changeover costs
 - Minimize disruption to System Operation Staff

High Availability / Disk Subsystem Level Backups

Production center



CAIXA Bank and Banco do Brasil

- CAIXA Bank
 - \$239 Billion Assets
 - 75,000 employees
- Banco do Brasil
 - \$466 Billion Assets
 - 109,000 employees

Challenge

- Convert major check processing application from Windows / Oracle to **Linux / PostgreSQL**
 - **Distributed among Workstations and Local Servers, and interconnected to Mainframe**

Convert Oracle to Postgres

- Convert Database
 - Schema
 - Data
 - Views
 - Triggers
 - Database Procedures
 - 50,000 lines of Oracle PL/SQL
 - Many lines are Oracle-specific syntax and semantics
 - Customer Application Interfaces to the Procedures

Converting Oracle Procedures (PL/SQL)

- **Many SQL statements are Oracle-specific**
 - Outer Join
 - Connect BY
 - Bulk Collect
- **Features not found in Postgres, given alternatives**
 - Packages
 - Global Variables
 - Indexed Arrays
- **Just Different**
 - Calling the Procedure
 - Returning results through Cursors
 - Exception handling

Challenge:

Create a tool to handle the conversion of Oracle Stored Procedures to PostgreSQL

The easy stuff – Examples of minor translations

- **SQL%NOTFOUND to NOT FOUND**
- **ROWNUM to limit**
- **SYSDATE to CLOCK_TIMESTAMP()**
- **FOR k IN REVERSE 1..j to FOR k IN REVERSE j..1**
- **ROWID to oid (beware of performance considerations)**
- **NVL to coalesce**
- **SQLCODE to global_util.sqlcode(SQLSTATE);**

Bulk Collect

```
CREATE OR REPLACE PROCEDURE allEmployees
IS
    TYPE v_array IS varray(50) OF char(14);
    arr_emp v_array;
BEGIN
    SELECT ename
        BULK COLLECT INTO arr_emp
    FROM emp
    ORDER BY ename;
    FOR i IN arr_emp.first..arr_emp.last LOOP
        dbms_output.put_line('|name'|arr_emp..(i)||'|i'| | i)
    END LOOP;
END allEmployees;
```

Oracle

```
CREATE OR REPLACE FUNCTION allEmployees()
RETURNS VOID AS $body$
DECLARE
    arr_emp char(14) [];
BEGIN
    arr_emp := array ( SELECT ename
        FROM emp
        ORDER BY ename);
    FOR i IN array_lower(arr_emp,1)..array_upper(arr_emp,1) LOOP
        RAISE NOTICE '|name % i %', arr_emp[i], i;
    END LOOP;
END; -- allEmployees
$body$ LANGUAGE plpgsql;
```

Postgres

Outer Join (Oracle)

```
CREATE or REPLACE PROCEDURE leftOuterJoin
AS
  TYPE artistInfo IS RECORD (
    lastName      char(20),
    firstName     char(20),
    title         varchar2(20));
  c_LOJ_row artistInfo; -- declare record
BEGIN
  DECLARE CURSOR c_LOJ IS
    SELECT lastName, firstName, NVL(title, '--') as title
    FROM artists tar, artists_albums tli, albums tal
    WHERE tar.artistId = tli.artistId
    and tli.albumId = tal.albumId(+)
    ORDER BY lastName, title;
  OPEN c_LOJ;
  LOOP
    FETCH c_LOJ INTO c_LOJ_row;
    EXIT WHEN c_LOJ%NOTFOUND;
  END LOOP;
  CLOSE c_LOJ;
END leftOuterJoin;
```

Outer Join (Postgres)

```
CREATE or REPLACE FUNCTION country.leftOuterJoin()
  RETURNS VOID AS $body$
DECLARE
  c_LOJ_row RECORD;
BEGIN
  DECLARE c_loj CURSOR IS
    SELECT lastName, firstName, coalesce(title, '--') as
      title
    FROM artists tar , artists_albums tli LEFT OUTER
      JOIN albums tal
      ON tli.albumId = tal.albumId
    WHERE tar.artistId = tli.artistId
    ORDER BY lastName , title;
  OPEN c_LOJ;
  LOOP
    FETCH c_LOJ INTO c_LOJ_row;
    IF NOT FOUND THEN
      EXIT;
    END IF;
  END LOOP;
  CLOSE c_LOJ;
END; -- leftOuterJoin
$body$ LANGUAGE plpgsql;
```

Calling the Stored Procedure from C

- The more challenging translations require more information than is available in the PL/SQL
- For example, **PL/pgSQL is tightly typed so function calls frequently need to have arguments cast**
- To achieve this, the DDL is inventoried (we use a repository)
- The translation also needs information regarding the input and output param of each function to correctly translate a function call
- This is achieved by preprocessing the PL/SQL source with a utility that inventories all functions and procedures and saves that information into the repository

Casting Function calls

The following illustrates the data type issue:

Consider the following table:

```
create table baseball.hittingStats (  
    teamId      smallint,  
    playerNum  smallint,  
    lastName   char(20),  
    atBats     numeric(4,0),  
    hits       numeric(4,0),  
    bb         numeric(3,0)  
);
```

Casting calls

Consider the following PL/SQL procedure:

```
CREATE OR REPLACE PROCEDURE sp_getStats (  
    p_teamId      IN hittingStats.teamId%TYPE,  
    p_playerNum   IN hittingStats.playerNum%TYPE,  
    v_avg         OUT NUMBER)  
  
IS  
  
BEGIN  
  
    SELECT..... .
```

This PL/SQL procedure is called in the following manner:

```
SQL> var myvar NUMBER;  
SQL> exec sp_getStats(1,6,:myvar);
```

Casting Function calls

When converted to pl/pgsql the result is:

```
CREATE OR REPLACE FUNCTION baseball.sp_getStats
(
    p_teamId      IN hittingStats.teamId%TYPE,
    p_playerNum   IN hittingStats.playerNum%TYPE,
    v_avg         OUT NUMERIC)

RETURNS NUMERIC AS $body$

DECLARE
BEGIN
    SELECT ..... . .
```

Casting Function calls

But when called in the following manner:

```
select v_avg as average from baseball.sp_getStats(1,6);
```

Postgres reports that it cannot find this function because it assumes that the args passed are integer. The error clearly reflects this:

```
ERROR:  function baseball.sp_getstats(integer, integer)
        does not exist
```

```
LINE 1: select v_avg as average from
        baseball.sp_getStats(1,6);
```

HINT: No function matches the given name and argument types. **You might need to add explicit type casts.**

Casting Function calls

The solution is to cast the pl/pgsql function call:

```
select v_avg as average from  
    baseball.sp_getStats(1::smallint, 6::smallint);
```

Not a trivial task if you have hundreds PL/SQL function calls to translate, many of which contain 10 or more args being passed

The toolset can help significantly with procedure call translation

Package translation

Use the accepted strategy of **generating a schema** for each package. Each procedure or function within the package is translated to a **pl/pgsql function** that is associated with the package.

Oracle example:

```
CREATE PACKAGE BODY baseball
IS
    PROCEDURE sp_scoutingReport (
        p_firstName    IN player.firstName%TYPE,
        p_lastName     IN player.lastName%TYPE,
        v_report       OUT varchar2) IS
...

```

Translates to Postgres:

```
CREATE SCHEMA baseball;
CREATE FUNCTION baseball.sp_scoutingReport (
    p_firstName IN player.firstName%TYPE,
    p_lastName  IN player.lastName%TYPE,
    v_report    OUT varchar )
RETURNS varchar AS $body$
...

```

Conclusion

- **Larger Customers have some unique technical requirements**
 - Mainframe
 - Cobol
 - Performance Details
 - Storage Subsystems
 - Trained Operational Staffs
 - More Code
- **Large Customers have vendor expectations**
 - Confidence to deal with them
 - Overall System Capability
 - Assurances of High Availability / Disaster Recovery
 - Detailed Approach / Plans for Conversion
 - Tools for Automation
 - Reference Accounts

Contacts

Ken.Rosensteel@bull.com

Dave.Edwards@bull.com



Architect of an Open World™

PGEast 2011

Package Global Variable translation

We manage global variables in the plperl `$_SHARED` hash which is global to the connection.

For example:

```
$_SHARED{$schema.$key}= $val;
```

Access Functions:

Store:

```
global_util.plSetGlobal(schema text, key text, val text);  
global_util.plSetGlobal(schema text, key text, val int);  
global_util.plSetGlobal(schema text, key text, val bigint);  
global_util.plSetGlobal(schema text, key text, val bool);
```

Retrieve:

```
global_util.plGetGlobal(schema text, key text);
```

Package Global Variable translation (cont)

Our tool operates like a compiler generating the required GET or SET call for each global encountered in a procedure or function.

Example:

Oracle PL/SQL:

```
v_index := 1;
```

Postgres:

```
v_index := 1;  
v_globalVarSet := global_util.plSetGlobal('schema', 'v_index', v_index::integer);
```

And define a local instance of the variable in the DECLARE section of the translated function.

Package Global Records and Arrays

Temp tables used to emulate PL/SQL global records and arrays in the translated plpgsql.

Code generated to create temp tables for global records and arrays.

PL/SQL access of these types translated to SQL that access the temp tables that emulate these types.

A PL/SQL global record will have a definition similar to the following in the package header:

```
TYPE r_hitter IS RECORD (playerNum hittingStats.playerNum%TYPE,  
                          ab        hittingStats.atBats%TYPE,  
                          hits      hittingStats.hits%TYPE,  
                          bb        hittingStats.bb%TYPE);
```

CREATE TABLE DDL for Postgres cannot have a clause such as table1.rank%TYPE

Inventoried DDL information used to generate a function that creates a temp table to contain the record.

Package Global Records and Arrays

Example of temp table generation for a global record:

```
CREATE FUNCTION global_util.gen_r_hitter_1()
  RETURNS void AS
  $$
BEGIN
  BEGIN
    TRUNCATE TABLE r_hitter_1;
  EXCEPTION
    when UNDEFINED_TABLE or WRONG_OBJECT_TYPE then
      BEGIN
        CREATE TEMP TABLE r_hitter_1(
          id                int NOT NULL UNIQUE,
          playerNum        smallint,
          ab                numeric(4),
          hits              numeric(3),
          bb                numeric(3));
      EXCEPTION
        when others then
          RAISE EXCEPTION 'create table r_hitter_1 issue NUM:%, DETAILS:%',
            SQLSTATE, SQLERRM;
        END;
      when others then
        RAISE EXCEPTION 'truncate table r_hitter_1 issue NUM:%, DETAILS:%',
          SQLSTATE, SQLERRM;
      END;
END; -- gen_r_hitter_1
$$ LANGUAGE plpgsql;
```

Package Global Records and Arrays

In PL/SQL a global array is an array of records declared in the package body. The following extends the record `r_hitter` into an associative indexed by array:

```
TYPE r_hitter IS RECORD (playerNum  hittingStats.playerNum%TYPE,  
                        ab          hittingStats.atBats%TYPE,  
                        hits        hittingStats.hits%TYPE,  
                        bb          hittingStats.bb%TYPE);
```

```
TYPE t_hitter is TABLE OF r_hitter INDEX BY PLS_INTEGER;
```

```
v_hitter t_hitter;
```

When declared in this manner in a package one has a global array that can be indexed in any manner the programmer desires. In particular, one can index this starting with 0 which is intuitive to c programmers.

Package Global Records and Arrays

A PL/SQL example of populating a global array (all players for a specific team):

```
CREATE OR REPLACE PACKAGE BODY baseball
IS
  TYPE r_hitter IS RECORD ( playerNum  hittingStats.playerNum%TYPE,
                           ab         hittingStats.atBats%TYPE,
                           hits       hittingStats.hits%TYPE,
                           bb         hittingStats.bb%TYPE);

  TYPE t_hitter is TABLE OF r_hitter INDEX BY PLS_INTEGER;
  v_hitter t_hitter;
  --
  PROCEDURE sp_getHitters(
    p_teamId  IN hittingStats.teamId%TYPE)
  IS
    CURSOR c_get_hitters is
      SELECT playerNum, atBats AS ab, hits, bb
      FROM hittingStats
      WHERE teamId = p_teamId
      ORDER BY playerNum;
    r_players r_hitter;
    v_index PLS_INTEGER;
  BEGIN
    v_hitter.DELETE;
    FOR r_players IN c_get_hitters LOOP
      v_index          := r_players.playerNum;
      v_hitter(v_index) := r_players;
      dbms_output.put_line('obtained hitter ' || r_players.playerNum);
    END LOOP;
  END sp_getHitters;
```

Package Global Records and Arrays

The translation result:

```
CREATE or REPLACE FUNCTION baseball.sp_getHitters(p_teamId IN
    hittingStats.teamId%TYPE)
    RETURNS VOID AS $body$
DECLARE
    c_get_hitters CURSOR is
        SELECT playerNum, atBats AS ab, hits, bb
            FROM hittingStats
            WHERE teamId = p_teamId ORDER BY playerNum;
    r_players RECORD;
    v_index integer;
BEGIN
    perform global_util.gen_t_hitter(); -- truncate or create table for
    v_hitter
    FOR r_players IN c_get_hitters LOOP
        v_index := r_players.playerNum; -- smallint to int.
        INSERT INTO t_hitter
            (id, playerNum, ab, hits, bb)
        VALUES
            ( v_index, r_players.playerNum, r_players.ab, r_players.hits,
            r_players.bb);
        RAISE NOTICE 'obtained hitter %', r_players.playerNum;
    END LOOP;

END; -- sp_getHitters
$body$ LANGUAGE plpgsql;
```