Part One: Java in the Database[1]

At the beginning was SQL, a high-level query language for relational databases. Then the need to extend SQL with procedural logic gave birth to the concept of stored procedures and their corresponding languages such as Oracle's PL/SQL. Stored procedures allow developing data logic that run in the database, decoupled from business and computational logic that run in the middle tier. However, the proprietary nature of stored procedure languages, leads to some concerns (or perception) of vendor lock-in and skills shortage. Java is a response to these concerns. The ANSI SQLJ Part-I specification defines "*SQL Routines and Types Using Java*". Although there are differences in their specific implementations, most RDBMS including Oracle, DB2, Sybase, even open-sources RDBMS such as PostGresSQL and to some extent MySQL, support Java as language for stored procedures and user-defined functions.

Chapter One discusses the rationale for stored procedures, the programming model and languages. Chapter Two tells you everything you ever wanted to know about the OracleJVM, its architecture, memory management, threading, class sharing techniques, the native Java compiler (NCOMP), security management, and contrasts it with the JDK VM. Chapter Three delves into the details of developing, deploying, and invoking Java applications in the database, including an extensive section on PL/SQL wrappers (a.k.a. Call Spec) for publishing Java (i.e., make it known) to SQL, and mapping SQL datatypes to/from Java/JDBC datatypes. Chapter Four describes atypical Java applications, which implement new database functionality using standard Java libraries. Finally, just for fun, in Chapter Five, you will run basic JACL, Jython, Scheme and Groovy scripts in the database, as proof of the concept of supporting non-Java languages in the database[2]. There is a growing adoption of Java in the database, among DBAs and database developers, after reading this book, you will probably become yourself an aficionado, if not already the case!

Chapter One: Stored Procedure as Database Programming Model

Although stored procedures have been around for more than a decade now, there still is a recurrent, almost ideological, debate on this programming model. Although it takes position in favor of stored procedures, the intent of this book is not to fuel this discussion, but elaborate on the benefits, assuming that there are situations where stored procedures are the right design choices. In this chapter, I will discuss the rationales for stored procedures, the obstacles to their adoption, languages used for writing stored procedures, and proprietary procedural languages such as PL/SQL versus open standards languages such as Java.

1.1 Rationale for Stored Procedures

As database developers and database administrators (DBAs) already know, stored procedures allow the exploitation of capabilities of relational database management systems (RDBMSs) to

[1] See Oracle JDBC, Oracle SQLJ and JPublisher in Part Two, then Database Web Services in Part Three, and Putting Everything Together in Part Four

[2] I MUST say that this proof of concept does not correspond to any Oracle product plan.

their fullest extent. The motivations to use stored procedures range from simplifying database programming to advanced data access to performance to centrally managed data logic and to optimizing network traffic.

Simplifying Database Programming

Procedural programming (also known as modular programming), as the name indicates, is based on the concepts of modules (also known as packages) and procedures (also known as functions, routines, subroutines, or methods). Each module consists of one or more procedures. The resulting code is simple and easier to read, debug, and maintain. Stored procedures are a mix of procedural code and SQL. The runtime of stored procedures is usually tightly integrated with the RDBMS but could also be loosely coupled, as an external runtime. Procedural languages include vendors' extensions to SQL such as PL/SQL, but also Basic/Visual Basic, COBOL, Pascal, C/C++, C#, Perl, and Java.

Centrally Managed Data Logic

By centralizing data logic, you can share it across all database projects, thereby avoiding code duplication and allowing flexible application development.

Avoids Code Duplication

Stored procedures are written once, centralized, and not dispersed across applications. When the procedure is updated, all consumer applications will automatically pick up the new version, at the next invocation.

Fosters Data Logic Sharing

Irrespective of their implementation language (e.g., proprietary, Java, 3GL), stored procedures are declared and known to the database catalog, through their SQL signature. In the Oracle case, this is achieved via a PL/SQL wrapper known as "Call Spec." Through this PL/SQL wrapper, , SQL, PL/SQL, Java in the database, thin clients (Web), rich clients (desktop), stand-alone Java , and middle-tier components,[3] access the same, centrally managed data logic. For example, a stored procedure can be used to send a notification email when a new order is placed or to invalidate the middle-tier cache to notify data change (see "Poor Man's Cache Invalidation" example in Chapter Four).

Creates Implementation Transparency

Interfaces allow effective modularization/encapsulation and shield consumers from implementation details, allowing multiple implementations. By decoupling the call interface (i.e., "Call Spec" in Oracle's vocabulary) from its actual implementation, the stored procedure may change over time from being written in PL/SQL to Java or the opposite, transparently to the requesters.

---

[3] *Mastering Enterprise JavaBeans*, 2nd edition, by Ed Roman, Scott W. Ambler, Tyler Jewell –(New York: John Wiley & Sons, 2002)

Performance: Run JDBC Applications Faster in the Database

Performance is one of the main motivations for using stored procedures. A few years ago, Oracle used PL/SQL stored procedures to optimize the performance of a benchmark version of the infamous J2EE Blueprints "PetStore"[4] application. This optimization prompted a heated debate in the Java/J2EE community. On the heels of this debate, Microsoft implemented and published the results of a .NET variant of the same benchmark, using—guess what?—stored procedures! The main criticism[5] was the lack of portability of PL/SQL or Transact SQL stored procedures across RDBMS. Well, this is precisely the raison d'etre of Java stored procedures.

The conclusion to derive from these experiences, as database programmers already know, is that stored procedures are the right design choice for efficient database programming. Stored procedures inherently incur minimal data movement, compared to a set of individual SQL statements that ship data outside the database. By processing data within the database (sorting, filtering) and returning just the results, stored procedures reduce network traffic and data movement. To cut to the chase, let's compare the performance of a Java application used as a stand-alone Java database connectivity (JDBC) application deployed on a Java development kit (JDK) virtual machine (VM) versus the same code deployed as a Java stored procedure running in the database (this is, by the way, an illustration of the claim that you can reuse existing Java/JDBC applications, with minor changes, in the database). The following example will already give you an overview of the few steps involved in creating, compiling, publishing, and executing Java in the database.

Setup

Configuration:

A Pentium 4 M 1.80-GHz laptop, with 1 GB of RAM using Windows XP Professional Version 2002, Oracle Database 10g R1, and the associated JDBC drivers.

Create a table with a Varchar2, BLOB, and CLOB columns, using the following script (in a SQL*Plus session):

```
SQL> connect scott/tiger;
SQL> drop table basic_lob_table;
SQL> create table basic_lob_table (x varchar2 (30), b blob, c clob);
```

The Java Aplication

```
Listing 1.1 -- TrimLob.java
===========================
/*
 * This sample shows basic BLOB/CLOB operations
 * It drops, creates, and populates table basic_lob_table
```

[4] http://www.oracle.com/technology/tech/java/oc4j/pdf/9ias_net_bench.pdf
[5] http://java.sun.com/blueprints/qanda/faq.html#stored_procedures

```
 * with columns of blob, clob data types in the database
 * Then fetches the rows and trim both LOB and CLOB
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

/*
 * You need to import the oracle.sql package to use
 * oracle.sql.BLOB
 */
import oracle.sql.*;

public class TrimLob
{
  public static void main (String args []) throws SQLException {
  Connection conn;
 /*
  * Where is your code running: in the database or outside?
  */
  if (System.getProperty("oracle.jserver.version") != null)
  {
  /*
   * You are in the database, already connected, use the default
   * connection
   */
  conn = DriverManager.getConnection("jdbc:default:connection:");
  }
  else
  {
  /*
   * You are not in the database, you need to connect to
   * the database
   */

   DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
   conn =
         DriverManager.getConnection("jdbc:oracle:thin:", "scott",
          "tiger");
  }
  long t0,t1;
   /*
    * Turn auto commit off
    * (Required when using SELECT FOR UPDATE)
    */
    conn.setAutoCommit (false);
    t0=System.currentTimeMillis();
    // Create a Statement
    Statement stmt = conn.createStatement ();
    // Make sure the table is empty
    stmt.execute("delete from basic_lob_table");
    stmt.execute("commit");

    // Populate the table
    stmt.execute ("insert into basic_lob_table values ('first', " +
                  "'0101010101010101010101010101', " +
                  "'one.two.three.four.five.six.seven')");
    stmt.execute ("insert into basic_lob_table values ('second', " +
                  "'02020202020202020202020202020202020202', " +
                  "'two.three.four.five.six.seven.eight.nine.ten')");

   /*
    * Retreive LOBs and update contents (trim); this can be done by doing
    * "select ... for update".
    */
    ResultSet rset = stmt.executeQuery
                          ("select * from basic_lob_table for update");
    while (rset.next ())
    {
      // Get the lobs
      BLOB blob = (BLOB) rset.getObject (2);
      CLOB clob = (CLOB) rset.getObject (3);
```

```
      // Show the original lengths of LOBs
      System.out.println ("Show the original lob length");
      System.out.println ("blob.length()="+blob.length());
      System.out.println ("clob.length()="+clob.length());

      // Truncate the lobs
      System.out.println ("Truncate LOBs to legnth = 6");
      blob.truncate (6);
      clob.truncate (6);

      // Show the lob length after truncate()
      System.out.println ("Show the lob length after truncate()");
      System.out.println ("blob.length()="+blob.length());
      System.out.println ("clob.length()="+clob.length());
    }

    // Close the ResultSet and Commit changes
    rset.close ();
    stmt.execute("commit");

    // Close the Statement
    stmt.close ();

    t1=System.currentTimeMillis();
    System.out.println ("====> Duration: "+(int)(t1-t0)+ "Milliseconds");
    // Close the connection
    conn.close ();
  }
}
```

Running the Java Application as a Stand-alone JDBC Application

Stand-alone JDBC applications run on JDK VM, against the database. For my test, the database, the JDBC driver and application, all run on the same machine. The following steps compile the Java class and execute it:

```
javac TrimLob.java
java –classpath %CLASSPATH% TrimLob
```

Running the Java Application as a Java Stored Procedure

```
TrimLobSp.sql (contains Java source and SQL commands)
=============

connect scott/tiger;
create or replace java source named TrimLob as

Rem
Rem Insert here the entire Trimlob.java used above
Rem

/

show errors;

alter java source TrimLob compile;

show errors;

create or replace procedure TrimLobSp as
 language java name 'TrimLob.main(java.lang.String[])';
/

show errors;
set serveroutput on
call dbms_java.set_output(50000);
```

```
call TrimLobSp();
```

The following table contrasts the performance of 10 invocations of the same Java code as stand-alone JDBC, and as Java stored procedure, on the same laptop, using exactly the same configuration (i.e., Oracle Database 10g Release 1 and its embedded OracleJVM):

Run#	Stand-alone JDBC	Java Stored Procedure
1st	570 ms	121 ms
2nd	241 ms	61 ms
3rd	240 ms	60 ms
4th	250 ms	50 ms
5th	230 ms	50 ms
6th	281 ms	50 ms
7th	280 ms	50 ms
8th	241 ms	50 ms
9th	250 ms	50 ms
10th	251 ms	50 ms

Although we cannot draw a universal conclusion, because of the elimination of network roundtrips and because it runs within the same address space as SQL, this JDBC application runs four to five times faster in the database than outside. This example proves that, when appropriate, you can move Java/JDBC applications to the database and run them faster.

Encapsulation

Encapsulation is an object-oriented design principle that lets you structure an application into modules that hide data structures from outside view and also protect it from unauthorized access. Stored procedures allow building specialized modules, which can be tuned by domain specialists and DBAs, shielding consumers from the arcane data structure and SQL programming. Encapsulation also hides differences between RDBMSs by presenting the same call interface over different enterprise information systems (see the TECSIS System Use case in part four of the book).

Security: Advanced Data Access Control

Database-related applications that have explicit knowledge of database schema login and password may compromise the security of your system and may break upon schema change. You can enforce security as part of your design by using JDBC data sources that remove and defer the actual database and login information to deployment time, and in addition, implement security policies in stored procedures (validate login information on each procedure call) and only allow users/apps to call these stored procedures. You can control database access use through customized, advanced, sophisticated data access logic that does CRUD (i.e., Create, Retrieve, Update, Delete) operations on tables while denying users direct access to these tables. Database

triggers are traditionally used to enforce referential integrity and constraints, thereby making sure that only valid data enters the database; stored procedures that implement more complex constraints, additional operational security restrictions (e.g., forbid salary table update during weekends!), can be implemented as triggers, on top of the built-in security mechanisms offered by the RDBMS engine.

Resource Optimization

All database clients accessing the same database schema run the same in-memory copy of the procedure, thereby reducing overall memory allocation. Also, as demoed above, depending on the level of integration, stored procedures can run within the same address space as the SQL engine, incurring minimal call overhead and optimizing memory utilization. In Chapter Two, I will describe in detail the internal mechanisms of the Java VM in the Oracle database.

Low-Cost Deployment

Independent software vendors (ISVs) and integrators already know that the ability to bundle their products on top of the database considerably simplifies installation, platforms support, and product distribution. Java integration with the database eliminates the needs for an external JDK/JRE and the headache of platform compatibility; furthermore, it works the same way on every platform on which the database runs.

Fully Utilize Database Capabilities

Part Four of this book describes how Oracle (*interMedia*), TECSIS Systems, Oracle Text, British Columbia CorporateOnline, and DBPrism CMS case studies use the database to its full extent.

1.2 Obstacles to the Adoption of Stored Procedures

The following concerns are usually invoked as showstoppers for adopting stored procedures: portability across database vendors, scalability, maintainability, and debug-ability. As discussed following, some of these concerns are valid, but others are misperceptions.

Lack of Portability across RDBMS Vendors

In corporate IT environments that use more than one RDBMS, DBAs and database developers have to learn different procedural languages such as PL/SQL, T-SQL, SQL/PL, and so on. Large IT organizations can afford to dedicate specific resources to each RDBMS for handling tasks ranging from managing databases to writing stored procedures. However, most organizations are looking for the flexibility of redeploying their staff of DBAs and developers according to their business priorities. Using Java across tiers and an RDBMS enables the portability of skills. Also, in the unlikely situation where an organization decides to move to a different RDBMS, it will have to migrate not only the schema and data but also the set of stored procedures developed over the years. Using Java leaves the door open for such a move because the Java sources and classes can be migrated smoothly, with minimal changes, losses, and cost.

Scalability

In typical application deployments, the clients (i.e., Web client, rich client, desktop) are deployed over the middle-tier nodes, which in turn funnel threads corresponding to clients against a pool of fewer database connections, typically an order of magnitude less than the number of concurrent clients/threads. Still, database scalability is crucial to middle-tier scalability. The session-based architecture of the Oracle database makes it scale linearly on symmetric multiprocessing (SMP) and quasi-linearly on clusters and grid nodes. To conclude, PL/SQL and Java stored procedures scale very well as far as the platform permits. In other words, the scalability of stored procedures is a by-product of the architecture of the target RDBMS and not a limitation of the stored procedure programming model, per se.

Maintenance and Resilience to Schema Change

Upon schema change (i.e., when changes to table/column names, locations, or references occur), the stored procedures need to be updated to adapt to the new schema; however, all of the applications built on top of those stored procedures remain unchanged and still return the exact result sets from the new database design. Shielding your applications (i.e., business logic) from the inevitable schema change by encapsulating the database schema within centralized stored procedures and validation logic is a small price to pay compared to the benefits of maintenance. Stored procedures act as interfaces between the data schema and the business logic layer, shielding each layer from changes occurring in the others. Encapsulation significantly reduces the ripple effect.

Hard to Debug

Most RDBMSs support stored procedures development and debugging through an integrated development environment (IDE) using either proprietary mechanisms such as Oracle's java.debugAgent (which falls into obsolete!) or standard mechanisms such as the Java Debug Wire Protocol (JDWP). Oracle JDeveloper integrates JDWP and allows simultaneous debugging of PL/SQL and Java stored procedures in the same session. Third-party IDE, which support JDWP, allow debugging PL/SQL and/or Java directly in the database. Alternatively, and this is what most Java developers do, you debug your Java code first outside the database (as JDBC application), and then deploy it in the database. The bottom line is that debugging stored procedures is more difficult than debugging middle-tier applications or presentation logic. You cannot just use your favorite development tool to debug stored procedures in the database, hence this legitimate issue.

Weak Support for Complex Types

This concern is rather a question of perception. As shown in Chapter Three, stored procedures can pass complex database types such as user-defined types (ADT), SQL object types, nested tables, VARRAY, and multilevel collections between the client program and the database. As illustrated by the SQLJ Object type example in Chapter Three, the standard SQLData interface allows custom mapping of user-defined types (ADT) in JDBC applications (i.e., stored procedures); furthermore, vendors' extensions such as the Oracle JDBC extensions allow

exchanging Oracle Object types between SQL (RDBMS) and JDBC applications (i.e., Java stored procedures).

1.3 Languages for Stored Procedures

This section discusses the pros and cons of using proprietary languages, Java, and the emerging category of .NET languages in the database.

Proprietary Languages

The following discussion applies to most proprietary languages for stored procedures;[6] however, I focus on the Oracle PL/SQL, which is widely used and regarded as one of the best vendor-supplied language for stored procedures.

Seamless Integration with SQL

Proprietary languages for stored procedure such as Oracle's PL/SQL are an extension to SQL and as such are well integrated into SQL with little or no data types conversion and optimized for faster SQL data access. PL/SQL is well suited for wrapping intensive SQL operations with moderately complex procedural logic.

IDE Support

Those languages benefit from a strong vendor-supplied development environment and also third-party IDE. As an example, the Oracle JDeveloper, as well as third-party IDE, provide nice environments for writing, debugging, and maintaining PL/SQL programs.

Portability

Cross-platform portability of proprietary language such as PL/SQL is inherited from the portability of the RDBMS. As an example, compiled PL/SQL packages can be moved to different platforms where the Oracle database runs—from Solaris to Linux or Windows or vice versa—without recompilation. Cross-vendor portability (e.g., run one vendor's language in another vendor's database) is technically possible (see CLR below) but not yet a sure thing.

Java for Stored Procedures

Complete Programming Language

The Java language is by design an object-oriented programming language that supports many programming models, including simple models such as JavaBean, POJO, JDBC applications, Java stored procedures, and more complex J2EE programming models such as Servlets, JavaServer Pages, and Enterprise Java Beans.

---

[6] Not including languages supported by Microsoft's Common Language Runtime such as Visual Basic and C#.

Secure Language

The Java language has built-in security mechanisms, such as the lack of pointer arithmetic, which prevents it from computing offending memory offset; the Garbage Collector, which reduces the risk of memory corruption by cleaning up dead objects and reclaiming unused memory; the type safety described hereafter; the byte-code verifier described later in this chapter; and Java 2 security for accessing system resources or remote systems (described in Chapter Two under the security section).

Type Safety

Java's strong typing[7] and static typing (i.e., compile time type checking) make the language less vulnerable to viruses and buffer overflow security holes. The creators of Java carefully designed the language and byte code formats to facilitate static type checking. The byte code verifier effectively checks static types at compile time, giving Java developers the opportunity to fix any type errors before deployment, resulting in a type safe program that runs efficiently.

Robustness

Java requires catching exceptions that can be thrown by methods in any class, thereby making Java stored procedures more robust. The automatic memory Garbage Collection also enforces robustness because it reduces the likelihood of memory corruption.

Productivity: Rapid Design Features

The Java language comes with a set of built-in rapid application design (RAD) features, such as the following:

- Built-in automatic bounds checking on arrays
- Built-in network access classes (java.net, java.rmi)
- Automatic Garbage Collector that eliminates whole classes of memory management issues
- Standard data types and application programming interfaces (APIs) contain many useful and ready-to-use classes

Using Java as a Procedural Language

Like most RDBMSs, the Oracle database promotes a simplified programming model that can be summarized as "no threading within applications code." Although OracleJVM lets you deploy a threaded Java code, its scheduler is nonpreemptive; in other words, the active thread will run until it is no longer run-able. The running Java application in a session is practically the only code running in the embedded Java VM. Java stored procedures also share the same simplicity with J2EE programming models: "no threading within components code"; the container itself is threaded, but the components (i.e., EJB, Servlet, JSP) are nonthreaded. Furthermore, Java experts discourage threading and recommend to only have very few, for applications robustness and portability, "Don't depend on the thread scheduler" [Bloch01]. This simplified programming

[7] Strong typing refers to the requirement that the type of each field and variable and the return type of each method be explicitly declared.

model also simplifies memory management, by removing the need to place memory allocation locks during garbage collection (GC).

Standard Specifications for Java Stored Procedures

The following American National Standards Institute (ANSI) specifications define SQLJ, Java stored procedures, and SQLJ Object types:

- *SQLJ Part 0*. "Database Language SQL—Part 10: Object Language Bindings (SQL/OLB)," ANSI X3.135.10-1998. Specifications for embedding SQL statements in Java methods. Similar to the traditional SQL facilities for embedded SQL in COBOL and C and other languages. The Java classes containing embedded SQL statements are precompiled to pure Java classes with JDBC calls. Also known as *SQL.*
- *SQLJ Part 1*. "SQL Routines using the Java Programming Language," ANSI NCITS N331.1. Specifications for installing Java classes in a SQL system and for invoking Java static methods as SQL stored procedures and functions. Also known as *Java stored procedures.*
- *SQLJ Part 2*. "SQL Types using the Java Programming Language," ANSI NCITS N331.2. Also known as *SQLJ Object Types.*

POJO-like Programming Model

What are POJOs? If you Google "Java POJO," you'll get the following definition.

> *POJO = "Plain Old Java Object." Term coined by Martin Fowler, RebeccaParsons, and Josh MacKenzie to denote a normal Java object that is not a JavaBean, an EntityBean, a SessionBean, etc., and does not serve any other special role or implement any special interfaces of any of the Java frameworks (EJB, JDBC, ...).*
>
> *Any Java object can run within an EJB container, but many people don't know that or forget it. Fowler et al. invented the acronym* POJO *so that such objects would have a "fancy name," thereby convincing people that they were worthy of use.*
>
> *POJOs are useful for creating a DomainModel. In contrast, the various types of beans and other special Java objects often have constraints that make it difficult to use them directly to model a domain.*

Stored procedures use explicit SQL statements through JDBC, and aren't therefore pure POJOs; however, they have in common the simplicity of their programming models. Unlike when using Enterprise JavaBeans (EJBs), you don't need to be a rocket scientist to get a Java stored procedure right. As a matter of fact, the next EJB specification (EJB 3.0) is looking at simplifying the EJB model by integrating the POJO programming model.

Stored Procedures and O/R Mapping

O/R Mapping generally refers to transparent mapping of Java objects to a relational database, which is achieved through several mechanisms (or programming models), including EJB CMP,

POJO, and Java Data Object (JDO)[8]. Stored procedures may be used by O/R mapping frameworks to perform a custom mapping of a specific object but are by no means a substitute. Stored procedures belong to explicit persistence mechanisms (i.e., SQL intrusive), whereas O/R mapping frameworks address transparent persistence (i.e., non-SQL intrusive).

Cross-Database Portability

Most RDBMSs (except SQL Server) support Java, either through a loosely coupled external JDK-based runtime or through a tight integration of the Java runtime with the database kernel (i.e., OracleJVM). Database developers who choose Java in the database motivate this choice, among other things, by its cross-vendor portability. Although Java stored procedures implementations differ from one vendor to another, Java is by far the most portable language for stored procedures. This book offers an in-depth covering of Oracle's implementation.

HUGE Class Library and Tools: Reduced Development Time and Costs

As we all know, the ability to reuse existing libraries results in quicker and lower-cost applications development. The availability of a rich and very large set of standard libraries as well as third-party class libraries is one of the biggest benefits that Java brings to database developers. The smart and lazy developers will extend their databases with new capabilities (see Chapter Four) in no time, writing only a few lines of code and scripts to adapt and glue Java with the database.

Skills Reuse

Because Java is one of the most dominant and widespread programming languages, it is likely that Java programmers already exist within your organization; furthermore, most new hires graduating from colleges have Java programming skills. The ability to use the same language across the middle tier (business/application logic) and the database tier (data logic) bolsters skills reuse, which in turn simplifies resource allocation, thereby reducing project costs. *Java Database Connectivity and SQL Data Access*

The OracleJVM embeds a special JDBC driver and SQLJ runtimes for SQL data access. This enables redeploying J2SE/JDBC/SQLJ applications in the database (see the "Run Your JDBC Applications Faster in the Database", above).

Starting with Java

An introduction to Java is beyond the scope of this book; however, here are some pointers to start with Java:

- Online Java Tutorial: http://java.sun.com/docs/books/tutorial
- The comp.lang.java FAQ List: http://www.ibiblio.org/javafaq/javafaq.html

[8] http://java.sun.com/products/jdo/

- The *Java Developer Almanac*

.NET Languages

SQL Server 2005 is said to introduce the Common Language Runtime (CLR) on top of the .NET framework, for running stored procedures written in C#, VB.NET, J#, and other languages. CLR can be viewed as a generic virtual machine, which supports multiple languages in the database. The most interesting aspect of CLR is its support by the latest releases of DB2 and Oracle; as a result, and similarly to Java, CLR allows the portability of code across the the Microsoft middle tier and database tier,[9] as well as the portability of code across RDBMSs.[10] Java is no longer the only portable language for stored procedures across RDBMSs but remains by far the most portable,[11] the most widely used, and offers the largest reusable set of code and class libraries. It is not yet clear what will be the uptake of C#, VB.NET beyond SQL Server 2005, because the version of CLRis not the same across vendors.

1.4 PL/SQL or Java

This is the $72,526 techno-political question being asked all the time: "*When should we use proprietary languages such as PL/SQL and when should we use open standard languages such as Java for stored procedures?"* The short but correct answer is, "*It depends!"* It depends on your goals, requirements such as the profile of the code being executed in the database (i.e., data access intensive versus computation intensive), the available skills within your organization, and whether you might need to migrate the code in question from the database to the middle tier or vice versa. According to a survey conducted by Evans Data Corporation among database developers (across all RDBMS), 22% declare using PL/SQL while 41% to 46% declare using Java (across all RDBMSs that support it). These figures are not exclusive; the person who declared using Java also uses PL/SQL when dealing with the Oracle Database. As you have already figured out, there is no straight answer; however, here are my own rules of thumb for choosing between Java and PL/SQL, but each DBA and database developer has his or her own rules or motivations for choosing one approach versus another:

- Prefer PL/SQL when (i) *your data logic (i.e., data processing or data validation logic) is SQL intensive, or* (ii) *you already have the skills*. Modeled after the ADA programming language, PL/SQL is an advanced procedural language. Its seamless integration with SQL and the Oracle database allows faster SQL data access with little or no type conversion. There is a large community with Oracle-supplied packages[12] and third-party libraries.
- Prefer Java in the database when (i) *your data logic has moderate SQL data access requirements (as opposed to SQL intensive) and moderate computational requirements (as opposed to compute intensive), or* (ii) *you already have the skills* (Java skills are more pervasive, most college graduates know Java), or (iii) *you need to accomplish things you cannot do in PL/SQL, such as interaction with ERP systems, RMI servers, Java/J2EE, and Web services*, or (iv) *you want to leave the door open for partitioning your application*

[9] Enabled by the Integration of .NET with the SQL Server 2005 RDBMS
[10] IBM DB2 release 8.2 and Oracle Database 10g release 2 support CLR 1.x.
[11] DB2, Oracle, Sybase, PortGreSQL, and MySQL
[12] http://www.oracle.com/technology/tech/pl_sql/index.html

between the middle tier and the database tier. There is a large Java community with tons of class libraries (standard and third party) that you can reuse. When your data logic becomes too complex, you just migrate it to the middle tier.

- Furthermore, you should consider Java/J2EE in the middle tier (stand-alone JDBC, JavaBeans, POJOs, EJBs, Servlets/Java ServerPages, and so on) when (i) *your business logic is complex or compute intensive with little to moderate direct SQL access, or* (ii) *you are implementing a middle-tier-driven presentation logic, or* (iii) *you require transparent Java persistence (i.e., POJOS, CMP EJB) as opposed to SQL intrusive persistence, or* (iv) *you require container-managed infrastructure services (transaction, security), or* (v) *many other reasons not specified here*. If your business logic becomes SQL data access bound, you may migrate it into the database tier. JavaBeans, POJOs, and J2EE design models may orthogonally use stored procedures (PL/SQL and/or Java) directly through JDBC or indirectly through O/R mapping frameworks.

If performance is the only motivation, when is Java in the database competitive because it is sandwiched between PL/SQL (SQL intensive) and Java/J2EE in the middle tier (compute intensive)? As illustrated in the previous section entitled "*Run JDBC Applications Faster in the Database*", when you combine Java and SQL, Java in the database wins as it incurs less data traffic, and minimal network over head.

> [A poster on Slashdot.org] "*While I agree that I would tend to abstract all SQL to some PL/SQL call that "DBAs who get it" have control over, there are LOTS of things that Java can do that are VERY handy, when viewed at in the application architecture point of view and not just in a SQL context*."

PL/SQL and Java!

Most database applications use PL/SQL and Java to glue together the rich set of database features. Pragmatic database developers use both Java and PL/SQL because these complement each other very well. Since Oracle9*i* release 2, the OracleJVM supports the standard Java Debugging Wire Protocol, which allows JDWP-enabled IDE such as the Oracle JDeveloper or third-party tools to debug both PL/SQL and Java within the same session.

Now that we have set the stage, in the next chapters, I'll walk you through the entrails of the Java runtime in the Oracle database, how to deploy your own Java classes, and real life use cases.