Foreign and Legacy Data

21 min. read · View original

Most of the world's useful data are either residing on a server beyond your control or inside a database management system other than Oracle. Either way, we refer to these data as *foreign* from the perspective of one's local Oracle database. Your objective is always the same: **Treat foreign data as though they were residing in local SQL tables.**

The benefits of physically or virtually dragging foreign data back to your Oracle cave are the following:

- New developers don't have to think about where data are coming from.
- Developers can work with foreign data using the same query language that they use every day: SQL.
- Developers can work with the same programming tools and systems that they've been using daily. They might use COBOL, they might use C, they might use Java, they might use Common Lisp, they might use Perl or Tcl, but you can be sure that they've learned how to send an SQL query to Oracle from these tools and therefore that they will be able to use the foreign data.

A good conceptual way to look at what we're trying to accomplish is the construction of an SQL view of the foreign data. A standard SQL view may hide a 5-way join from the novice programmer. A *foreign table* hides the fact that data are coming from a foreign source.

We will refer to the system that makes foreign data available locally as an *aggregation architecture*. In designing an individual aggregation architecture you will need to address the following issues:

1. Coherency: Is it acceptable for your local version of the data to be out of sync with the foreign data? If so, to what extent?
2. Updatability: Is your local view updatable? I.e., can a programmer perform a transaction on the foreign database management system by updating the local view?
3. Social: To what extent is the foreign data provider willing to let you into his or her database?

Degenerate Aggregation

If the foreign data are stored in an Oracle database and the people who run that database are cooperative, you can practice a degenerate form of aggregation: Oracle to Oracle communication.

The most degenerate form of degenerate aggregation is when the foreign data are in the same Oracle installation but owned by a different user. To make it concrete, let's assume that you work for Eli Lilly in the data warehousing department, which is part of marketing. Your Oracle user name is "marketing". The foreign data in which you're interested are in the `prozac_orders` table, owned by the "sales" user. In this case your aggregation architecture is the following:

1. connect to Oracle as the foreign data owner (sales) and GRANT SELECT ON PROZAC_ORDERS TO MARKETING
2. connect to Oracle as the local user (marketing) and type SELECT * FROM SALES.PROZAC_ORDERS

Done.

Slightly Less Degenerate Aggregation

Imagine now that Prozac orders are processed on a dedicated on-line transaction processing (OLTP) database installation and your data warehouse (DW) is running on a physically separate computer. Both the OLTP and DW systems are running the Oracle database server and they are connected via a network. You need to take the following steps:

1. set up SQL*Net (Net8) on the OLTP system
2. set up the DW Oracle server to be a client to the OLTP server. If you are not using the Oracle naming services, you must edit $ORACLE_HOME/network/admin/tnsnames.ora to reference the OLTP system.
3. create a "marketing" user on the OLTP system
4. on the OLTP system, log in as "sales" and GRANT SELECT ON PROZAC_ORDERS TO MARKETING
5. on the DW system, create a database link to the OLTP system named "OLTP": CREATE DATABASE LINK OLTP CONNECT TO MARKETING IDENTIFIED BY *password for marketing* USING 'OLTP';
6. on the DW system, log in as "marketing" and SELECT * FROM SALES.PROZAC_ORDERS@OLTP;

In both of these degenerate cases, there were no coherency issues. The foreign data were queried in real-time from their canonical database. This was made possible because of social agreement. The owners of the foreign data were willing to grant you unlimited access. Similarly, social issues decided the issue of updatability. With a GRANT only on SELECT, the foreign table would not be updatable.

Non-Oracle-Oracle Aggregation

What if foreign data aren't stored in an Oracle database or they are but you can't convince the owners to give you access? You will need some sort of computer program that knows how to fetch some or all of the foreign data and stuff it into an Oracle table locally. Let's start with a concrete example.

Suppose you work at Silicon Valley Blue Cross. The dimensional data warehouse has revealed a strong correlation between stock market troubles and clinical depression. People working for newly public companies with volatile stocks tend to get into a funk when their paper wealth proves illusory. The suits at Blue Cross think that they can save money and doctors' visits by automatically issuing prescriptions for Prozac whenever an insured's employer's stock drops more than 10% in a day. To find candidates for happy pills, the following query should suffice:

```
select patients.first_names, patients.last_name,
stock_quotes.percent_change
from patients, employers, stock_quotes
where patients.employer_id = employers.employer_id
and employers.ticker_symbol =
stock_quotes.ticker_symbol
and stock_quotes.percent_change < -0.10
order by stock_quotes.percent_change
```

The `stock_quotes` table is the foreign table here. Blue Cross does not operate a stock exchange. Therefore the authoritative price change data must necessarily be pulled from an external source. Imagine that the external source is http://quote.yahoo.com/. The mature engineering perspective on a Web site such as quote.yahoo.com is that it is an object whose methods are its URLs and the arguments to those methods are the form variables. To get a quotation for the software company Ariba (ticker ARBA), for example, we need to visit http://quote.yahoo.com/q?s=arba in a Web browser. This is invoking the method "q" with an argument of "arba" for the form variable "s". The results come back in a human-readable HTML page with a lot of presentation markup and English text. It would be more convenient if Yahoo gave us results in a machine-readable form, e.g., a comma-separated list of values or an XML document. However, the HTML page may still be used as long as its structure does not vary from quote to quote and the percent change number can be pulled out with a regular expression or other computer program.

What are the issues in designing an aggregation architecture for this problem? First is coherency. It would be nice to have up-to-the-minute stock quotes but, on the other hand, it seems kind of silly to repeatedly query quote.yahoo.com for the same symbol. In fact, after 4:30 PM eastern time when the US stock market closes, there really isn't any reason to ask for a new quote on a symbol until 9:30 AM the next day. Given some reasonable assumptions about caching, once the `stock_quotes` table has been used a few times, queries will be able to execute much much faster since quote data will be pulled from a local cache rather than fetched over the Internet.

We don't have to think very hard about updatability. Blue Cross does not run a stock exchange therefore Blue Cross cannot update a stock's price. Our local view will not be updatable.

The social issue seems straightforward at first. Yahoo is making quotes available to any client on the public Internet. It looks at first glance as though our computer program can only request one quote at a time. However, if we fetch http://quote.yahoo.com/q?s=arba**+ibm**, we can get two quotes at the same time. It might even be possible to grab all of our insureds' employers' stock prices in one big page. A potential fly in the ointment is Yahoo's terms of service at http://docs.yahoo.com/info/terms/ where they stipulate

```
10. NO RESALE OF SERVICE

You agree not to reproduce, duplicate, copy, sell,
resell or exploit for
any commercial purposes, any portion of the Service,
use of the Service,
or access to the Service.
```

Where and when to run our programs

We need to write a computer program (the "fetcher") that can fetch the HTML page from Yahoo, pull out the price change figures, and stuff them into the `stock_quotes` table. We also need a more general computer program (the "checker") that can look at the foreign data required, see how old the cached data in `stock_quotes` are, and run the fetcher program if necessary.

There are three Turing-complete computer languages built into Oracle: C, Java, PL/SQL. "Turing-complete" means that any program that can be written for any computer can be written to run inside Oracle. Since you eventually want the foreign data to be combined with data inside Oracle, it makes sense to run all of your aggregation code inside the database. Oracle includes built-in functions to facilitate the retrieval of Web pages (see http://oradoc.photo.net/ora816/server.816/a76936/utl_http.htm#998100).

In an ideal world you could define a database trigger that would fire every time a query was about to SELECT from the `stock_quotes` table. This trigger would somehow figure out which rows of the foreign table were going to be required. It would run the checker program to make sure that none of the cached data were too old, and the checker in turn might run the fetcher.

Why won't this work? As of Oracle version 8.1.6, it is impossible to define a trigger on SELECT. Even if you could, there is no advertised way for the triggered program to explore the SQL query that is being executed or to ask the SQL optimizer which rows will be required.

The PostgreSQL RDBMS has a "rule system" (see http://www.postgresql.org/docs/programmer/x968.htm) which can intercept and transform a SELECT. It takes the output of the SQL parser, applies one or more transformation rules, and produces a new set of queries to be executed. For example, a rule may specify that any SELECT which targets the table "foo" should be turned into a SELECT from the table "bar" instead; this is how Postgres implements views. As it stands, the only transformation that can be applied to a SELECT is to replace it with a single, alternative SELECT - but PostgreSQL is open source software which anyone is free to enhance.

The long term fix is to wait for the RDBMS vendors to augment their products. Help is on the way. The good news is that a portion of the ANSI/ISO SQL-99 standard mandates that RDBMS vendors, including Oracle, provide support for wrapping external data sources. The bad news is that the SQL-99 standard is being released in chunks, the wrapper extension won't be published until 2001, and it may be several years before commercial RDBMSes implement the new standard.

The short term fix is to run a procedure right before we send the query to Oracle:

```
call checker.stock_quotes( 0.5 )

select patients.first_names, patients.last_name,
stock_quotes.percent_change ...
```

Our checker is an Oracle stored procedure named `checker.stock_quotes`. It checks every ticker symbol in `stock_quotes` and calls the fetcher if the quote is older than the specified interval, measured in days. If we want to add a new `ticker_symbol` to the table, we call a different version of `checker.stock_quotes`:

```
call checker.stock_quotes( 0.5, 'IBM' )
```

If there is no entry for IBM which is less than half a day old, the checker will ask the fetcher to get a stock quote for IBM.

An Aggregation Example

Blue Cross will dispense a lot of Prozac before Oracle implements the SQL-99 wrapper extension. So let's build a `stock_quotes` foreign table which uses Java stored procedures to do the checking and fetching. We'll begin with a data model:

```
create table stock_quotes (
        ticker_symbol           varchar(20) primary
key,
        last_trade              number,
        -- the time when the last trade occurred
(reported by Yahoo)
        last_trade_time         date,
        percent_change          number,
        -- the time when we pulled this data from Yahoo
        last_modified           date not null
);
```

This is a stripped-down version, where we only store the most recent price quote for each ticker symbol. In a real application we would certainly want to maintain an archive of old quotes, perhaps by using triggers to populate an audit table whenever `stock_quotes` is updated. Even if your external source provides its own historical records, fetching them is bound to be slower, less reliable, and more complicated than pulling data from your own audit tables.

We'll create a single source code file, `StockUpdater.java`. Oracle 8.1.6 includes a Java compiler as well as a virtual machine, so when this file is ready we can load it into Oracle and compile it with a single command:

```
bash-2.03$ loadjava -user username/password -resolve -
force StockUpdater.java
    ORA-29535: source requires recompilation
    StockUpdater:171: Class Perl5Util not found.
    StockUpdater:171: Class Perl5Util not found.
    StockUpdater:218: Class PatternMatcherInput not
found.
    StockUpdater:218: Class PatternMatcherInput not
found.
     Info: 4 errors
loadjava: 6 errors
bash-2.03$
```

Oops. The `-resolve` option tells Oracle's `loadjava` utility to compile and link the class right away, but `StockUpdater` depends on classes that haven't yet been loaded into Oracle. Most Java virtual machines are designed to automatically locate and load classes at runtime by searching through the filesystem, but the Oracle JVM requires every class to be loaded into the database in advance.

We need to obtain the `Perl5Util` and `PatternMatcherInput` classes. These are part of the Oro library, an open-source regular expression library that's available from http://jakarta.apache.org/oro/index.html. When we download and untar the distribution, we'll find a JAR file that contains the classes we need. We'll load the entire JAR file into Oracle and then try to load `StockUpdater` again.

```
bash-2.03$ loadjava -user username/password -resolve
jakarta-oro-2.0.jar
bash-2.03$ loadjava -user username/password -resolve -
force StockUpdater.java
bash-2.03$
```

These commands take a while to execute. When they're done, we can check the results by running this SQL query:

```
SELECT RPAD(object_name,31) ||
       RPAD(object_type,14) ||
       RPAD(status,8)
       "Java User Objects"
  FROM user_objects
 WHERE object_type LIKE 'JAVA %';
```

Here's a small portion of the output from this query:

```
Java User Objects
----------------------------------------------------
StockUpdater                   JAVA CLASS    VALID
StockUpdater                   JAVA SOURCE   VALID
org/apache/oro/text/awk/OrNode JAVA CLASS    VALID
org/apache/oro/text/regex/Util JAVA CLASS    VALID
org/apache/oro/util/Cache      JAVA CLASS    VALID
org/apache/oro/util/CacheFIFO  JAVA CLASS    VALID
...
```

Our source code is marked `VALID`, and there's an associated class which is also `VALID`. There are a bunch of `VALID` regexp classes. All is well.

If we wanted, we could have compiled the `StockUpdater` class using a free-standing Java compiler and then loaded the resulting class files into Oracle. We aren't required to use the built-in Oracle compiler.

The `-force` option forces `loadjava` to overwrite any existing class with the same name, so if we change our class we don't necessarily have to drop the old version before loading the new one. If we do want to drop one of Oracle's stored Java classes, we can use the `dropjava` utility.

Calling our Stored Procedures

> **Figure 15-1**: The aggregation architecture. The client application obtains data by querying Oracle tables using SQL. To keep the foreign tables up to date, the application calls the Checker and the Fetcher, which are Java stored procedures running inside Oracle. The Checker is called via two layers of PL/SQL: one layer is a call spec which translates a PL/SQL call to a Java call, and the other is a wrapper procedure which provides an autonomous transaction for the aggregation code to run in.

In order to call Java stored procedures from SQL, we need to define *call specs*, which are PL/SQL front ends to static Java methods. Here's an example of a call spec:

```
PROCEDURE stock_quotes_spec ( interval IN number )
    AS LANGUAGE JAVA
    NAME 'StockUpdater.checkAll( double )';
```

This code says: "when the programmer calls this PL/SQL procedure, call the `checkAll` method of the Java class `StockUpdater`." The `checkAll` method must be `static`: Oracle doesn't automatically construct a new `StockUpdater` object.

We don't allow developers to use the call spec directly. Instead we make them call a separate PL/SQL procedure which initiates an *autonomous transaction*. We need to do this because an application might call the checker in the middle of a big transaction. The checker uses the fetcher to add fresh data to the `stock_quotes` table. Now the question arises: when do we commit the changes to the `stock_quotes` table? There are three options:

1. Have the fetcher issue a COMMIT. This will commit the changes to `stock_quotes`. It will also commit any changes that were made before the checker was called. This is a very bad idea.

2. Have the fetcher update `stock_quotes` without issuing a COMMIT. This is also a bad idea: if the calling routine decides to abort the transaction, the new stock quote data will be lost.

3. Run the checker and the fetcher in an independent transaction of their own. The fetcher can commit or roll back changes without affecting the main transaction. Oracle provides the `AUTONOMOUS_TRANSACTION` pragma for this purpose, but the pragma doesn't work in a call spec - it's only available for regular PL/SQL procedures. So we need a separate layer of glue code just to initiate the autonomous transaction.

Here's the SQL which defines all of our PL/SQL procedures:

```
CREATE OR REPLACE PACKAGE checker AS
    PROCEDURE stock_quotes( interval IN number );
    PROCEDURE stock_quotes( interval IN number,
ticker_symbol IN varchar );
END checker;
/
show errors



CREATE OR REPLACE PACKAGE BODY checker AS


    -- Autonomous transaction wrappers
    PROCEDURE stock_quotes ( interval IN number )
    IS
        PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
        stock_quotes_spec( interval );
    END;

    PROCEDURE stock_quotes ( interval IN number,
ticker_symbol IN varchar )
    IS
        PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
        stock_quotes_spec( interval, ticker_symbol );
    END;


    -- Call specs
    PROCEDURE stock_quotes_spec ( interval IN number )
        AS LANGUAGE JAVA
        NAME 'StockUpdater.checkAll( double )';

    PROCEDURE stock_quotes_spec ( interval IN number,
ticker_symbol IN varchar )
        AS LANGUAGE JAVA
        NAME 'StockUpdater.checkOne( double,
java.lang.String )';


END checker;
/
show errors
```

We've placed the routines in a *package* called `checker`. Packages allow us to group procedures and datatypes together. We're using one here because packaged procedure definitions can be *overloaded*. The `checker.stock_quotes` procedure can be called with either one or two arguments and a different version will be run in each case. The `stock_quotes_spec` procedure also comes in two versions.

Writing a Checker in Java

We're ready to start looking at the `StockUpdater.java` file itself. It begins in typical Java fashion:

```
// Standard Java2 classes, already included in Oracle
import java.sql.*;
import java.util.*;
import java.io.*;
import java.net.*;

// Regular expression classes
import org.apache.oro.text.perl.*;
import org.apache.oro.text.regex.*;



public class StockUpdater {
```

Then we have the two checker routine, starting with the one that updates the entire table:

```
    public static void checkAll( double interval )
    throws SQLException {

        // Query the database for the ticker symbols
that haven't
        // been updated recently
        String sql = new String( "SELECT ticker_symbol
" +
                                 "FROM stock_quotes " +
                                 "WHERE (sysdate -
last_modified) > " +
                                    String.valueOf(
interval ) );

        // Build a Java List of the ticker symbols

        // Use JDBC to execute the given SQL query.
        Connection conn = getConnection();
        Statement stmt = conn.createStatement();
        stmt.execute( sql );
        ResultSet res = stmt.getResultSet();

        // Go through each row of the result set and
accumulate a list
        List tickerList = new ArrayList();
        while ( res.next() ) {
            String symbol = res.getString(
"ticker_symbol" );
            if ( symbol != null ) {
                tickerList.add( symbol );
            }
        }
        stmt.close();

        System.out.println( "Found a list of " +
tickerList.size() + " symbols.");

        // Pass the List of symbols on to the fetcher
        fetchList( tickerList );
    }
```

This routine uses JDBC to access the `stock_quotes` table. JDBC calls throw exceptions of type `SQLException` which we don't bother to catch; instead, we propagate them back to the calling programmer to indicate that something went wrong. We also print debugging information to standard output. When running this class outside Oracle, the debug messages will appear on the screen. Inside Oracle, we can view them by issuing some SQL*Plus commands in advance:

```
SET SERVEROUTPUT ON
CALL dbms_java.set_output(5000);
```

Standard output will be echoed to the screen, `5000` characters at a time.

The second checker operates on one ticker symbol at a time, and is used to add a new ticker symbol to the table:

```
    public static void checkOne( double interval,
String tickerSymbol )
    throws SQLException {

        // Set up a list in case we need it
        List tickerList = new ArrayList();
        tickerList.add( tickerSymbol );

        // Query the database to see if there's recent
data for this tickerSymbol
        String sql = new String( "SELECT " +
                                 "  ticker_symbol, " +
                                 "  (sysdate -
last_modified) as staleness " +
                                 "FROM stock_quotes " +
                                 "WHERE ticker_symbol =
'" + tickerSymbol + "'");
        Connection conn = getConnection();
        Statement stmt = conn.createStatement();
        stmt.execute( sql );
        ResultSet res = stmt.getResultSet();

        if ( res.next() ) {
            // A row came back, so the ticker is in the
DB
            // Is the data recent?
            if ( res.getDouble("staleness") > interval
) {
                // Fetch fresh data
                fetchList( tickerList );
            }

        } else {
            // The stock isn't in the database yet
            // Insert a blank entry
            stmt.executeUpdate( "INSERT INTO
stock_quotes " +
                                "(ticker_symbol,
last_modified) VALUES " +
                                "('" + tickerSymbol +
"', sysdate)" );
            conn.commit();

            // Now refresh the blank entry to turn it
into a real entry
            fetchList( tickerList );
        }
        stmt.close();
    }
```

Writing a Fetcher in Java

The fetcher is implemented as a long Java method called `fetchList`. It begins by retrieving a Web page from Yahoo. For speed and simplicity, we extract all of the stock quotes on a single page.

```
    /** Accepts a list of stock tickers and retrieves
stock quotes from Yahoo Finance
        at http://quote.yahoo.com/
    */
    private static void fetchList( List tickerList )
    throws SQLException {
        // We need to pass Yahoo a string containing
ticker symbols separated by "+"
        String tickerListStr = joinList( tickerList,
"+" );

        if ( tickerListStr.length() == 0 ) {
            // We don't bother to fetch a page if there
are no ticker symbols
            System.out.println("Fetcher: no ticker
symbols were supplied");
            return;
        }

        try {
            // Go get the Web page
            String url = "http://quote.yahoo.com/q?s="
+ tickerListStr;
            String yahooPage = getPage( url );
```

The fetcher uses a helper routine called `getPage` to retrieve Yahoo's HTML, which we stuff into the `yahooPage` variable. Now we can use Perl 5 regular expressions to extract the values we need. We create a new `Perl5Util` object and use the `split()` and `match()` methods to extract the section of the page where the data is:

```
            // ... continuing the definition of
fetchList ...

            // Get a regular expression matcher
            Perl5Util regexp = new Perl5Util();

            // Break the page into sections using
</table> tags as boundaries
            Vector allSections = regexp.split(
"/<\\/table>/", yahooPage );

            // Pick out the section which contains the
word "Symbol"
            String dataSection = "";
            boolean foundSymbolP = false;
            Iterator iter = allSections.iterator();
            while ( iter.hasNext() ) {
                dataSection = (String) iter.next();
                if ( regexp.match( "/<th.*?
>Symbol<\\/th>/", dataSection )) {
```

```
                    foundSymbolP = true;
                    break;
                }
            }

            // If we didn't find the section we wanted,
throw an error
            if ( ! foundSymbolP ) {
                throw new SQLException( "Couldn't find
the word 'Symbol' in " + url );
            }
```

We need to pick out today's date from the page. This is the date when the page was retrieved, which we'll call the "fetch date". Each stock quote also has an individual timestamp, which we'll call the "quote date". We use a little class of our own (`OracleDate`) to represent dates, and a helper routine (`matchFetchDate`) to do the regexp matching.

```
            OracleDate fetchDate = matchFetchDate(
dataSection );
            if ( fetchDate == null ) {
                throw new SQLException("Couldn't find
the date in " + url);
            }
            System.out.println("The date appears to be:
'" + fetchDate.getDate() + "'");
```

If we can't match the fetch date, we throw an exception to tell the client programmer that the fetcher didn't work. Perhaps the network is down, or Yahoo's server is broken, or Yahoo's graphic designers decided to redesign the page layout.

We're ready to extract the stock quotes themselves. They're in an HTML table, with one row for each quote. We set up a single JDBC statement which will be executed over and over, using placeholders to represent the data:

```
            String update_sql = "UPDATE stock_quotes
SET " +
                "last_trade = ?, " +
                "last_trade_time = to_date(?, ?), " +
                "percent_change = ?, " +
                "last_modified = sysdate " +
                "WHERE ticker_symbol = ? ";
            Connection conn = getConnection();
            PreparedStatement stmt =
conn.prepareStatement( update_sql );
```

Now we pick apart the HTML table one row at a time, using a huge regexp that represents an entire table row. By using a `PatternMatcherInput` object, we can make `regexp.match()` traverse the `dataSection` string and return one match after another until it runs out of matches. For each stock quote we find, we clean up the data and perform a database INSERT.

```
            // Use a special object to make the regexp
search run repeatedly
            PatternMatcherInput matchInput = new
PatternMatcherInput( dataSection );

            // Search for one table row after another
            while ( regexp.match( "/<tr.*?>.*?" +
                                  "<td nowrap.*?>(.*?)
<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)
<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)
<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)
<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)
<\\/td>.*?" +
                                  "<\\/tr>/s" ,
matchInput )) {
                // Save the regexp groups into
variables
                String tickerSymbol = regexp.group(1);
                String timeStr = regexp.group(2);
                String lastTrade = regexp.group(3);
                String percentChange = regexp.group(5);

                // Filter the HTML from the ticker
symbol
                tickerSymbol =
regexp.substitute("s/<.*?>//g", tickerSymbol);
                stmt.setString( 5, tickerSymbol );

                // Parse the time stamp
                OracleDate quoteDate = matchQuoteDate(
timeStr, fetchDate );
```

```
                if ( quoteDate == null ) {
                    throw new SQLException("Bad date
format");
                }
                stmt.setString( 2, quoteDate.getDate()
);
                stmt.setString( 3,
quoteDate.getDateFormat() );

                // Parse the lastTrade value, which may
be a fraction
                stmt.setFloat( 1, parseFraction(
lastTrade ));

                // Filter HTML out of percentChange,
and remove the % sign
                percentChange = regexp.substitute(
"s/<.*?>//g", percentChange);
                percentChange = regexp.substitute(
"s/%//g", percentChange);
                stmt.setFloat( 4, Float.parseFloat(
percentChange ));

                // Do the database update
                stmt.execute();
            }

            stmt.close();
            // Commit the changes to the database
            conn.commit();

        } catch ( Exception e ) {
            throw new SQLException( e.toString() );
        }
    } // End of the fetchList method
```

Helper Routines for the Fetcher

The fetcher loads HTML pages using the `getPage` method, which uses the `URL` class from the Java standard library. For a simple HTTP GET, this routine is all we need.

```
    /** Fetch the text of a Web page using HTTP GET
     */
    private static String getPage( String urlString )
    throws MalformedURLException, IOException {
        URL url = new URL( urlString );
        BufferedReader pageReader =
            new BufferedReader( new InputStreamReader(
url.openStream() ) );
        String oneLine;
        String page = new String();
        while ( (oneLine = pageReader.readLine()) !=
null ) {
            page += oneLine + "\n";
        }
        return page;
    }
```

Dates, along with their Oracle format strings, are stored inside `OracleDate` objects. `OracleDate` is an "inner class", defined inside the StockUpdater class. Because it is a private class, it can't be seen or used outside of StockUpdater. Later, if we think `OracleDate` will be useful for other programmers, we can turn it into a public class by moving the definition to a file of its own.

```
    /** A class which represents Oracle timestamps. */
    private static class OracleDate {
        /** A string representation of the date */
        private String date;
        /** The date format, in Oracle's notation */
        private String dateFormat;

        /** Methods for accessing the date and the
format */
        String getDate() { return date; }
        String getDateFormat() { return dateFormat; }
        void setDate( String newDate ) { date =
newDate; }
        void setDateFormat( String newFormat ) {
dateFormat = newFormat; }

        /** A constructor that builds a new OracleDate
*/
        OracleDate( String newDate, String newFormat )
{
```

```
            setDate( newDate );
            setDateFormat( newFormat );
        }
    }
```

To extract the dates from the Web page, we have a couple of routines called `matchFetchDate` and `matchQuoteDate`:

```
    /** Search through text from a Yahoo quote page to
find a date stamp */
    private static OracleDate matchFetchDate( String
text ) {
        Perl5Util regexp = new Perl5Util();

        if ( regexp.match("/<p>\\s*
(\\S+\\s+\\S+\\s+\\d+\\s+\\d\\d\\d\\d)\\s+[0-9:]+[aApP]
[mM][^<]*<table>/", text) ) {
            return new OracleDate( regexp.group(1),
"Day, Month DD YYYY" );

        } else {
            return null;
        }
    }



    /** Search through the time column from a single
Yahoo stock quote
        and set the time accordingly.  */
    private static OracleDate matchQuoteDate( String
timeText, OracleDate fetchDate ) {
        Perl5Util regexp = new Perl5Util();

        if ( regexp.match("/\\d?\\d:\\d\\d[aApP][mM]/",
timeText) ) {
            // When the time column of the stock quote
doesn't include the day,
            // the day is pulled from the given
fetchDate.
            String date = fetchDate.getDate() + " " +
timeText;
            String format = fetchDate.getDateFormat() +
" HH:MIam";
            return new OracleDate( date, format );
        } else if ( regexp.match("/[A-Za-z]+
+\\d\\d?/", timeText )) {
            // After midnight but before the market
opens, Yahoo reports the date
            // rather than the time.
            return new OracleDate( timeText, "Mon DD"
);

        } else {
            return null;
        }
    }
```

The stock prices coming back from Yahoo often contain fractions, which have special HTML markup. The `parseFraction` method pulls the HTML apart and returns the stock price as a Java `float`:

```
    /** Convert some HTML from the Yahoo quotes page to
a float, handling
        fractions if necessary */
    private static float parseFraction( String s ) {
        Perl5Util regexp = new Perl5Util();
        if ( regexp.match( "/^\\D+(\\d+)\\s*<sup>(\\d*)
</sup>/<sub>(\\d*)</sub>/", s)) {
            // There's a fraction
            float whole_num = Float.parseFloat(
regexp.group(1) );
            float numerator = Float.parseFloat(
regexp.group(2) );
            float denominator = Float.parseFloat(
regexp.group(3) );
            return whole_num + numerator / denominator;

        } else {
            // There is no fraction
            // strip the HTML and go
            return Float.parseFloat( regexp.substitute(
"s/<.*?>//g", s) );
        }
    }
```

Odds and Ends

All of our methods obtain their JDBC connections by calling `getConnection()`. By routing all database connection requests through this method, our class will be able to run either inside or outside the database - `getConnection` checks its environment and sets up the connection accordingly. Loading Java into Oracle is a tedious process, so it's nice to be able to debug your code from an external JVM.

```
    public static Connection getConnection()
    throws SQLException {

        Connection conn;

        // In a real program all of these constants
should
        // be pulled from a properties file:
        String driverClass =
"oracle.jdbc.driver.OracleDriver";
        String connectString =
"jdbc:oracle:oci8:@ora8i_ipc";
        String databaseUser = "username";
        String databasePassword = "password";

        try {
            // Figure out what environment we're
running in
            if (
System.getProperty("oracle.jserver.version") == null )
{
                // We're not running inside Oracle
                DriverManager.registerDriver(
(java.sql.Driver)
Class.forName(driverClass).newInstance() );
                conn = DriverManager.getConnection(
connectString, databaseUser, databasePassword );

            } else {
                // We're running inside Oracle
                conn = DriverManager.getConnection(
"jdbc:default:connection:" );
            }

            // The Oracle JVM automatically has
autocommit=false,
            // and we want to be consistent with this
if we're in an external JVM
            conn.setAutoCommit( false );
            return conn;

        } catch ( Exception e ) {
            throw new SQLException( e.toString() );
        }
    }
```

To call `StockUpdater` from the command line, we also need to provide a `main` method:

```
    /** This method allows us to call the class from
the command line
     */
    public static void main(String[] args)
    throws SQLException {
        if ( args.length == 1 ) {
            checkAll( Double.parseDouble( args[0] ));
        } else if ( args.length == 2 ) {
            checkOne( Double.parseDouble(args[0]),
args[1] );
        } else {
            System.out.println("Usage: java
StockUpdater update_interval [stock_ticker]");
        }
    }
```

Finally, the fetcher needs a utility which can join strings together. It's similar to the "join" command in Perl or Tcl.

```
    /** Builds a single string by taking a list of
strings
        and sticking them together with the given
separator.
        If any of the elements of the list is not a
String,
        an empty string in inserted in place of that
element.
    */
    public static String joinList( List stringList,
String separator ) {

        StringBuffer joinedStr = new StringBuffer();

        Iterator iter = stringList.iterator();
        boolean firstItemP = true;
        while ( iter.hasNext() ) {
            if ( firstItemP ) {
                firstItemP = false;
            } else {
                joinedStr.append( separator );
            }

            Object s = iter.next();
            if ( s != null && s instanceof String ) {
                joinedStr.append( (String) s );
            }
        }
        return joinedStr.toString();
    }
} // End of the StockUpdater class
```

A Foreign Table In Action

Now that we've implemented all of this, we can take a look at our foreign table in SQL*Plus:

```
SQL> select ticker_symbol, last_trade,
     (sysdate - last_modified)*24 as hours_old
     from stock_quotes;

TICKER_SYMBOL        LAST_TRADE  HOURS_OLD
-------------------- ---------- ----------
AAPL                    22.9375 3.62694444
IBM                     112.438 3.62694444
MSFT                      55.25 3.62694444
```

This data is over three hours old. Let's request data from within the last hour:

```
SQL> call checker.stock_quotes( 1/24 );

Call completed.

SQL> select ticker_symbol, last_trade,
     (sysdate - last_modified)*24 as hours_old
     from stock_quotes;

TICKER_SYMBOL        LAST_TRADE  HOURS_OLD
-------------------- ---------- ----------
AAPL                     23.625 .016666667
IBM                     114.375 .016666667
MSFT                    55.4375 .016666667
```

That's better. But I'm curious about the Intel Corporation.

```
SQL> call checker.stock_quotes( 1/24, 'INTC' );

Call completed.

SQL> select ticker_symbol, last_trade,
     (sysdate - last_modified)*24 as hours_old
     from stock_quotes;

TICKER_SYMBOL         LAST_TRADE  HOURS_OLD
-------------------- ---------- ----------
AAPL                     23.625 .156666667
IBM                     114.375 .156666667
MSFT                    55.4375 .156666667
INTC                         42 .002777778
```

Reference

Next: Normalization

---

mbooth@arsdigita.com
philg@mit.edu

Reader's Comments

> Okay, I think the statute of limitations on this page has expired, so I'm free to point out that (alas!) my email address is no longer mbooth@arsdigita.com.
>
> You can find me at michaelfbooth.com.
>
> -- Michael Booth, November 15, 2007

Add a comment | Add a link