Apéndice A

CLIs
[*Call-Level Interfaces*]

1. Introducción 2

2. SQL [Structured Query Language] 3

3. ODBC [Open DataBase Connectivity] de MicroSoft 8

4. BDE [Borland Database Engine] de Borland 9

5. JDBC [Java DataBase Connectivity] 10

6. Oracle 17

6.1 PL/SQL 18

6.2 Pro*C (Embedded SQL) 22

7. Estudio Comparativo 23

7.1 Pruebas 25

7.2 Conclusiones 32

8. Referencias Bibliográficas 33

1. Introducción

"... existen tantos criterios importantes pero conflictivos
que su reconciliación y solución adecuadas
constituyen una gran tarea de ingeniería
que demanda del diseñador [del lenguaje] un profundo
conocimiento de todos los aspectos del arte de programar..."

C. A. R. Hoare
Hints on Programming Language Design (1973)

En la actualidad existe una gran cantidad de lenguajes de programación, tanto específicos como de propósito general. Cada lenguaje resulta adecuado para determinadas tareas. A su vez, para cada uno de estos lenguajes, el programador suele tener distintas alternativas para acceder a bases de datos desde sus programas.

Se denomina ***CLI*** [*Call-Level Interface*] a un protocolo estándar de acceso a una base de datos desde un lenguaje de alto nivel. Casi todas las empresas que comercializan compiladores intentan imponer su propio estándar: ODBC de Microsoft, BDE de Borland... Por su parte, las entidades que desarrollan servidores de bases de datos también recomiendan interfaces diseñadas por ellos (como Oracle). Finalmente, también existen algunos estándares que no son propiedad de una empresa y, en teoría, deberían funcionar con productos de procedencia dispar (este es el caso de JDBC).

La mayor parte de los *CLI* existentes en el mercado (prácticamente todos los que manejan bases de datos relacionales) emplean SQL a la hora de realizar consultas sobre la base de datos, por lo que su conocimiento previo es esencial para cualquier programador interesado en realizar aplicaciones que involucren operaciones sobre bases de datos relacionales.

Es este estudio se intentarán exponer y comparar distintas alternativas utilizadas actualmente en el desarrollo de aplicaciones que involucran operaciones sobre bases de datos (generalmente relacionales). En concreto, se analizarán dos interfaces utilizadas en Windows (ODBC de Microsoft y BDE de Borland), un paquete estándar del lenguaje Java (JDBC) y dos de las alternativas ofrecidas por Oracle Corporation (PL/SQL y Pro*C).

2. SQL

"El modelo relacional representa el lenguaje ensamblador de los sistemas modernos (y futuros) de bases de datos"

C. J. Date

El artículo "A Relational Model of Data for large Shared Data Banks" fue publicado por E. F. Codd en Communications of the ACM en junio de 1970. El modelo relacional de Codd es la base de los sistemas de gestión de bases de datos relacionales [*RDBMSs: Relational DataBase Management Systems*].

El lenguaje "*Structured English Query Language*" (SEQUEL) fue desarrollado por IBM Corporation haciendo uso del modelo de Codd. Posteriormente, SEQUEL se convirtió en SQL [*Structured Query Language*]. En 1979, Relational Software Inc. (hoy Oracle Corporation) introdujo en el mercado la primera versión comercial de SQL. El último estándar de SQL publicado por ANSI e ISO se denomina SQL-92 (a veces, SQL2): ANSI X3.135–1992 e ISO/IEC 9075:1992.

DDL [Data Definition Language]

Las órdenes DDL de SQL incluyen aquéllas que nos permiten manipular tablas y vistas. A continuación se describen, tal como se utilizan en Oracle, los comandos ofrecidos por SQL para manipular tablas (crearlas, modificarlas y eliminarlas).

Creación de tablas

```
CREATE TABLE [schema.]table
     ( { column datatype [DEFAULT expr]
       [ column_constraint ] ...
       [ table_constraint] }
    [, { column datatype [DEFAULT expr]
       [ column_constraint ] ...
       [ table_constraint ] }...)
```

Valores por defecto:

DEFAULT sirve para asociarle un valor por defecto a una columna (se evalúa al crear la tabla, no cada vez que se inserte una tupla):

```
· DEFAULT NULL
· DEFAULT USER (nombre de usuario)
· DEFAULT literal
```

Restricciones de integridad:

1. Columnas

```
[CONSTRAINT constraint]
      { [NOT] NULL
      | UNIQUE
      | PRIMARY KEY
      | REFERENCES [schema.]table [(column)] [ON DELETE CASCADE]
      | CHECK (condition)}
```

2. Tablas

```
[CONSTRAINT constraint]
      { { UNIQUE|PRIMARY KEY} (column [,column] ...)
      | FOREIGN KEY (column [,column] ...) REFERENCES [schema.]table
      [(column [,column] ...)] [ON DELETE CASCADE]
      | CHECK (condition)}
```

Aclaraciones:

· NULL: Cuando una columna puede contener valores nulos (por defecto).

· NOT NULL: Cuando una columna no puede contener valores nulos.

· UNIQUE: La columna determina una clave (pueden aparecer varias tuplas con NULL pero no varias tuplas con el mismo valor si éste es distinto de NULL).

· PRIMARY KEY: Columna o conjunto de columnas que determinan la clave primaria de la tabla (no pueden aparecer valores nulos).

· FOREIGN KEY: Clave externa (restricción de integridad referencial).

· REFERENCES: Clave PRIMARY o UNIQUE referenciada por FOREIGN KEY. Por defecto será la clave primaria de la tabla referenciada.

· ON DELETE CASCADE: Mantenimiento automático de la integridad referencial al eliminar el valor referenciado.

· CHECK: Especificación de una condición que deben satisfacer las tuplas de la tabla.

Modificación de tablas

```
ALTER TABLE [schema.]table
  ADD ({[COLUMN] column datatype [DEFAULT expr] [column_constraint]
      [, [COLUMN] column datatype [DEFAULT expr] [column_constraint] ...]
        |table_constraint})

ALTER TABLE [schema.]table
   DROP {[COLUMN] column [RESTRICT|CASCADE]
         |drop_clause}
```

Explicaciones:

· ADD: Se utiliza para añadir columnas o restricciones de integridad a la tabla.
· DROP: Permite eliminar columnas o restricciones de integridad de la tabla.

· RESTRICT: Si hay vistas que incluyen la columna, la columna no se elimina.
· CASCADE: Todas las vistas dependientes se actualizan recursivamente.

Eliminación de tablas

```
DROP TABLE [schema.]table [CASCADE | CASCADE CONSTRAINTS | RESTRICT]
```

Anotaciones:

· CASCADE: Elimina automáticamente vistas y restricciones de integridad referencial.
· CASCADE CONSTRAINTS: Elimina automáticamente restricciones de integridad referencial.
· RESTRICT: No se elimina la tabla si existen vistas o restricciones de integridad referencial.

DML [Data Manipulation Language]

Para completar la exposición de lo que nos permite hacer SQL a continuación se describe la sintaxis de las órdenes SELECT, INSERT, DELETE y UPDATE, las cuatro órdenes existentes en SQL para manipular las tuplas contenidas en las tablas de una base de datos.

Consultas

```
SELECT [DISTINCT | ALL]
  { * | { [schema.]{table | view}.* | expr [c_alias] }
     [, { [schema.]{table | view}.* | expr [c_alias] } ] ... }
  FROM [schema.]{table | view} [t_alias]
  [WHERE condition]
  [[START WITH condition ] CONNECT BY condition]
  [GROUP BY expr [, expr] ... [HAVING condition] ]
  [{UNION | UNION ALL | MINUS} SELECT command ]
  [ORDER BY {expr|position} [ASC|DESC][, {expr|position} [ASC|DESC]] ...]
  [FOR UPDATE [OF [[schema.]{table | view}.]column
               [, [[schema.]{table | view}.]column] ...]]
```

Control de duplicados:
· DISTINCT: Devuelve sólo una copia de los duplicados.
· ALL: Devuelve todas las copias (por defecto).

Selección de columnas:
· *: Todas las columnas de las tablas y vistas incluidas en la claúsula FROM.
· table.*: Todas las columnas de la tabla.
· view.*: Todas las columnas de la vista.

Condiciones de la consulta:
· WHERE: Condición de la consulta
· CONNECT BY y START WITH: Consultas jerárquicas (operador PRIOR y pseudocolumna LEVEL).

Formato de la salida:
· GROUP BY y HAVING: Agrupar tuplas.
· ORDER BY: Salida ordenada (ASC por defecto)

Acceso concurrente:
· FOR UPDATE: Bloquea las tuplas seleccionadas.

Consultas "avanzadas":
· UNION: Devuelve la unión de las tuplas resultado de las consultas.
· UNION ALL: Como UNION, además incluye los duplicados.
· MINUS: Devuelve las tuplas de la primera consulta no obtenidas por la segunda.

Inserción de tuplas

```
INSERT INTO [schema.]{table | view}  [ (column [, column] ...) ]
  { VALUES (expr [, expr] ...) | subquery }
```

☛ schema: el que contiene la tabla o la vista sobre la que se realiza la inserción.
☛ table: nombre de la tabla donde se desea insertar tuplas.
☛ view: nombre de la vista en cuya tabla base se desea insertar tuplas.
☛ column: columna de la tabla o de la vista.

· VALUES: Especificación de una tupla que ha de insertarse en la tabla (o en la vista).
· *subquery*: Sentencia SELECT cuyo resultado de la consulta se inserta en la tabla.

Si se omite alguna columna de la tabla en la especificación (column [, column]) , ésta será inicializada a su valor por defecto (el especificado al crear la tabla). Si no se especifican las columnas involucradas en la inserción, la claúsula VALUES o la consulta asociada a la inserción deben establecer los valores de todas las columnas de la tabla.

El número de columnas especificadas debe coincidir con el número de valores proporcionados. Si no se especifica en qué columnas debe realizarse la inserción, el número de valores proporcionados debe coincidir con el grado de la tabla (su número de columnas).

Eliminación de tuplas

```
DELETE FROM [schema.]{table | view}
  [WHERE condition]
```

☛ schema: el que contiene la tabla o la vista sobre la que se realiza el borrado de tuplas.
☛ table: nombre de la tabla de la que se desea eliminar tuplas.
☛ view: nombre de la vista de cuya tabla base se desea eliminar tuplas.

· WHERE: Claúsula que especifica la condición que deben satisfacer las tuplas para ser borradas. Si no se especifica una claúsula WHERE, se eliminan todas las tuplas de la tabla.

Actualización de tuplas

```
UPDATE [schema.]{table | view} [alias]
   SET { column = expr  } [, column = expr ] ...
   [WHERE condition]
```

- ☛ schema: el que contiene la tabla o la vista sobre la que se realiza la actualización.
- ☛ table: nombre de la tabla que se desea actualizar.
- ☛ view: nombre de la vista cuya tabla base se desea actualizar.
- ☛ alias: Renombra la tabla (o vista) para las demás claúsulas de la orden UPDATE.
- ☛ column: nombre de la columna que debe actualizarse

· SET: Indica los valores actualizados que deben tomar las distintas columnas.
· WHERE: Restringe la actualización a aquellas tuplas que verifican la condición.

Si se omite alguna columna, su valor no cambia. Si no se especifica una claúsula WHERE, se actualizan todas las tuplas de la tabla (o vista).

Control de transacciones

Para completar la funcionalidad de SQL sólo faltan por describir los comandos COMMIT y ROLLBACK que permiten gestionar transacciones de una forma cómoda:

Commit

```
COMMIT [WORK]
```

Da por finalizada la transacción actual. Los cambios producidos en la base de datos se hacen permanentes.

Rollback

```
ROLLBACK [WORK]
```

Deshace los cambios efectuados en la última transacción.

NOTA ACLARATIVA:

WORK es opcional y no tiene ningún efecto en Oracle. Fue incluido para cumplir con el estándar ANSI de SQL.

3. ODBC

ODBC [Open DataBase Connectivity] define un protocolo abierto para realizar conexiones a servidores de bases de datos. Para acceder a un servidor, la aplicación debe usar el API de ODBC definido en el estándar. Además, las sentencias SQL deben cumplir la sintaxis de ODBC SQL.

Según Microsoft, el acceso a bases de datos a través de ODBC no es más lento, ODBC no limita la funcionalidad para obtener portabilidad y, ante todo, ODBC simplifica el trabajo del programador.

Los controladores para las bases de datos se incluyen en DLLs [*Dynamically Linked Libraries*]. Estos drivers traducen las peticiones ODBC a peticiones interpretables por el servidor de bases de datos. Usando distintas DLLs se puede acceder simultáneamente a distintos servidores (DB/2, SQL Server, Oracle, Informix...). ODBC también permite acceder a ficheros locales (dBASE DBF, Excel XLS, Access MDB, etc.). En este último caso, el propio controlador hace las veces de servidor.

En una aplicación que haga uso de ODBC se distinguen cuatro componentes arquitectónicos principales:

① **Fuente de información [*Data Source*]**: Definido en una entrada del fichero ODBC.INI o en el registro de MS Windows, que puede incluir los ficheros en los que está almacenada la información (p.ej. ficheros locales de tipo dBASE) o información relativa al DBMS [*DataBase Management System*] del que se debe obtener la información.

② **Controlador [*Driver*]**: DLL que procesa las llamadas ODBC para adaptarlas a la fuente de la que se obtiene información. Es la interfaz entre la aplicación ODBC y el DBMS particular que se utiliza.

③ **Gestor de controladores [*Driver Manager*]**: DLL suministrada por Microsoft como parte del ODBC SDK [*ODBC Software Development Kit*]. Carga los controladores en memoria y despacha las llamadas a funciones del ODBC API.

④ **Aplicación**: El programa que aprovecha la funcionalidad ofrecida por ODBC. Generalmente será necesario algún compilador de Microsoft (Visual Basic o Visual C++) y se ejecutará en Windows.

4. BDE

BDE (acrónimo correspondiente a "Borland Database Engine") es la alternativa a ODBC utilizada por los productos de Borland: Delphi®, Delphi Client/Server®, IntraBuilder®, C++ Builder®, Paradox® for Windows y Visual dBASE® for Windows. El API de BDE se denominaba antiguamente IDAPI [*Integrated Database Application Program Interface*].

BDE facilita la programación de aplicaciones cliente/servidor que accedan a bases de datos remotas (DB2, InterBase, Oracle, Sybase...) o a ficheros locales (Paradox, dBASE, FoxPro, Access, etc.). Entre las cualidades destacables de BDE destaca que, si no disponemos de un controlador BDE para la base de datos a la que hemos de acceder, BDE permite utilizar los drivers ODBC.

BDE se caracteriza por tener un ***API orientado a objetos*** (ODBC no) que nos permite programar a muy alto nivel el acceso a múltiples bases de datos. En tiempo de ejecución se crean objetos para manipular tablas y consultas SQL. Como sucedía en ODBC, BDE no se ciñe completamente al estándar SQL-92 (para permitir el acceso a ficheros locales Paradox y dBASE combinado con el acceso a servidores SQL remotos a través de *"Borland's Local SQL"*).

El núcleo de BDE está formado por un conjunto de DLLs que gestionan el acceso a las bases de datos (equivalente al *DriverManager* de ODBC): DBCLIENT.DLL, IDPROV.DLL, IDDR32.DLL, IDAPI32.DLL, BLW32.DLL (soporte multinacional, para lo que utiliza los ficheros BLL), IDBAT32.DLL (batch operations), IDQBE32.DLL (Query-By-Example), IDSQL32.DLL (SQL) e IDR20009.DLL (mensajes de error).

Entre los controladores disponibles para el acceso a ficheros locales se encuentran: IDASCI32.DLL (ficheros ASCII), IDPDX32.DLL (Paradox), IDDBAS32.DLL (dBASE) e IDDAO32.DLL (Access). El fichero IDODBC32.DLL es el controlador que permite acceder a bases de datos a través de drivers ODBC.

Borland suministra un programa (BDE Administrator, BDEADMIN.EXE) para controlar la configuración de BDE en el registro de Windows y los alias del fichero IDAPI.CFG, el núcleo de todo el sistema BDE.

A pesar de que el API de BDE sea más cómodo de utilizar que el de ODBC, lo usual a la hora de programar es utilizar los componentes (como *TQuery* o *TTable*) disponibles en los compiladores visuales de Borland (*Delphi* o *C++Builder*).

5. JDBC

"Que sea sencillo: lo más sencillo posible, pero no más"

Albert Einstein

JDBC [*Java DataBase Connectivity*] es la interfaz suministrada con el JDK [*Java Development Kit*] que permite acceder de manera uniforme a distintos servidores de bases de datos (a todos aquellos para los que exista el correspondiente controlador JDBC). JDBC no es más que un conjunto de clases, interfaces y excepciones que posibilitan la comunicación, mediante SQL, de una aplicación o applet Java con, potencialmente, cualquier DBMS.

Contenido del paquete java.sql

Interfaces

• CallableStatement	*Llamada a procedimientos almacenados*
• Connection	*Conexión a un servidor*
• DatabaseMetaData	*Catálogo del DBMS*
• Driver	
• PreparedStatement	*Sentencia SQL precompilada*
• ResultSet	*Resultado de una consulta*
• ResultSetMetaData	*Metainformación de ResultSet*
• Statement	*Sentencia SQL*

Clases

- Date
- DriverManager
- DriverPropertyInfo
- Time
- Timestamp
- Types

Excepciones

• DataTruncation	
• SQLException	*Error en el uso de SQL*
• SQLWarning	*Aviso (posible error al usar SQL)*

Uso de JDBC

En todos los programas que utilicen JDBC se deben realizar cuatro operaciones básicas: importar el paquete *java.sql* (para poder utilizar las clases definidas en el estándar), cargar el driver JDBC propio del DBMS que se vaya a utilizar (para poder acceder a la base de datos), establecer una conexión y liberar la conexión tras realizar las operaciones deseadas.

☞ *Utilización del paquete java.sql*

El primer paso necesario para aprovechar lo que JDBC nos ofrece es, lógicamente, importar el paquete en el que están definidas las clases, interfaces y excepciones de JDBC:

```
import java.sql.*;
```

☞ *Carga del driver JDBC*

Para acceder a un servidor concreto es estrictamente necesario disponer del controlador JDBC adecuado para el DBMS particular(en este caso distintas versiones de Oracle). El driver, que será un fichero .jar [Java Archive] o .zip, deberá encontrarse en el directorio de la aplicación o en el *CLASSPATH*.

Para utilizar Personal Oracle Lite es necesario el fichero *POLJDBC.JAR*. Para cargar el driver tendremos que escribir:

```
try {
    Class.forName("oracle.pol.poljdbc.POLJDBCDriver");
} catch (Exception e) {
    System.out.println(e);
    System.exit(0);
}
```

Si nuestro servidor es Oracle 8, necesitaremos el fichero *CLASSES102.ZIP* o bien *CLASSES111.ZIP*, según estemos utilizando la versión 1.0.2 del JDK o la versión 1.1. Además, bajo Windows, será necesaria una DLL (que ha de encontrarse en el *PATH*). En el caso de Oracle 8, la librería de enlace dinámico es *OCI803JDBC.DLL* (o una versión posterior) y el código necesario para cargar el controlador JDBC es el siguiente:

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (Exception e) {
    System.out.println(e);
    System.exit(0);
}
```

☞ *Establecimiento de una conexión*

Una vez registrado el driver (cargado llamando al método *Class.forName*), el siguiente paso necesario es establecer una conexión con el DBMS. Para ello necesitaremos un objeto que implemente el interfaz definido por *Connection*:

```
Connection conexion = null;
```

La conexión se puede establecer de distintas formas, siendo en todas necesario especificar a qué base de datos nos queremos conectar. Además, debemos fijar el protocolo que se utilizará al conectarnos a la base de datos. Toda esta información está recogida en una URL [Uniform Resource Locator] de la forma ***jdbc:protocolo:[nombreBD][:@localización]***.

vg:	*Oracle 7*	*URL = "jdbc:oracle:"*
	Oracle Lite	*URL = "jdbc:POLite:POLite"*
	Oracle 8	*URL = "jdbc:oracle:oci8:@SERVIDOR"*

El *DriverManager*, que es la interfaz entre la aplicación y los controladores JDBC cargados, se encargará de encontrar el controlador capaz de realizar la conexión con la base de datos especificada.

A la hora de establecer la conexión disponemos de distintas alternativas, siendo las dos siguientes las más usuales:

Primera alternativa:
Automáticamente se nos pedirá el login y el password para establecer la conexión

```
try {
    conexion = DriverManager.getConnection(URL);
} catch (Exception e) {
    ...
}
```

Segunda alternativa:
Directamente especificamos tanto el login como el password

```
try {
    conexion = DriverManager.getConnection (URL, usuario, password);
} catch (Exception e) {
    ...
}
```

☞ *Liberación de una conexión*
Para liberar una conexión simplemente invocamos al método *close( )* de la clase *Connection*:

```
try {
    conexion.close();
} catch (Exception e) {
    ...
}
```

Obtención de metainformación en JDBC

JDBC permite obtener metainformación de la base de datos a través de las clases *DatabaseMetaData* y *ResultSetMetaData*. Los siguientes ejemplos están realizados para el driver JDBC de Oracle Lite y de hecho no funcionan para otros drivers JDBC:

DATABASEMETADATA: *Consulta al catálogo del sistema*

```
// NB: POLite ---> 'DatabaseMetaData.getCatalogs' not supported
//             ---> 'DatabaseMetaData.getSchemas' -> NAME

ResultSet tablas = null;

try {
      DatabaseMetaData metadata = conexion.getMetaData();

      tablas = metadata.getTables(null,null,"%",null);

      // ResultSetMetaData aux = tablas.getMetaData();
      //
      // aux.getColumnCount() —> 5
      //
      // aux.getColumnName(i) -> TABLE_QUALIFIER (1), TABLE_OWNER (2)
      //                 TABLE_NAME (3), TABLE_TYPE (4) y REMARKS (5)

      while (tablas.next())
            System.out.println(tablas.getString("Table_Name"));

} catch (SQLException e) ...
```

RESULTSETMETADATA: *Información acerca de una tabla concreta*

```
ResultSet columnas = null;

try {
      columnas =
            conexion.getMetaData().getColumns(null,null,"TABLA","%");

      // ResultSetMetaData aux = columnas.getMetaData();

      // aux.getColumnCount()  —> Número de columnas (12)
      // aux.getColumnName(i)  -> Descripción

      // Campos disponibles: TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME,
      // COLUMN_NAME, DATA_TYPE, TYPE_NAME, PRECISION, LENGTH, SCALE,
      // RADIX, NULLABLE y REMARKS

      while (columnas.next())
            System.out.println( columnas.getString("Column_Name")
                  + "\t" + columnas.getString("Type_Name")
                  + "\t" + columnas.getShort("Data_Type") );

} catch (SQLException e) ...
```

Uso de SQL a través de JDBC

JDBC proporciona tres clases (interfaces para ser más precisos) que permiten utilizar SQL en nuestras aplicaciones escritas en Java:

✐ STATEMENT

Usada para sentencias simples SQL. Para crear un objeto de esta clase es necesario disponer de una conexión abierta y ejecutar el método *Connection.createStatement( )*.

✐ PREPAREDSTATEMENT

Subclase de *Statement* usada para sentencias SQL simples que se ejecuten frecuentemente y para sentencias SQL con parámetros. En teoría su ejecución es más eficiente ya que la consulta se precompila. Un objeto de este tipo se crea llamando al método *Connection.prepareStatement*.

✐ CALLABLESTATEMENT

Finalmente, esta subclase de *PreparedStatement* se utiliza para llamar a procedimientos almacenados en la base de datos [*stored procedures*]. Los objetos de esta clase se crean invocando al método *Connection.prepareCall*.

Ejemplos:

CREATE TABLE

```
Statement SQLcreate = null;

try {
      SQLcreate = conexion.createStatement();

      SQLcreate.execute("CREATE TABLE EMPLEADO
                        (ID INTEGER, NAME CHAR(30), JOB CHAR(30))");
      conexion.commit();

} catch (SQLException e) ...
```

INSERT

```
Statement SQLinsert = null;

try {
      SQLinsert = conexion.createStatement();

      SQLinsert.execute ( "INSERT INTO EMPLEADO"
                        + " VALUES (1, 'Frank', 'Student')");
      conexion.commit();

} catch (SQLException e) ...
```

```
Statement SQLselect = null;
ResultSet resultado = null;

try {
      SQLselect = conexion.createStatement();
} catch (SQLException e) {
      System.out.println("Error: " + e + "\n");
      System.exit(0);
}

// select * from empleado

try {
      resultado = SQLselect.executeQuery("SELECT * FROM EMPLEADO");

      while (resultado.next()) {

          int    getID   = resultado.getInt("id");
          String getName = resultado.getString("name");
          String getJob  = resultado.getString("job");

          // Equivalente a:
          // int    getID   = resultado.getInt(1);
          // String getName = resultado.getString(2);
          // String getJob  = resultado.getString(3);
          ...
      }

} catch (SQLException e) {

      while(e != null) {
            System.out.println(e.getMessage());
            e = e.getNextException();
      }

}

// select distinct JOB from empleado

try {
      resultado = SQLselect.executeQuery
                                ("select distinct JOB from EMPLEADO");

      while (resultado.next())
            System.out.println(resultado.getString("job"));

} catch (SQLException e) {

      while(e != null) {
            System.out.println(e.getMessage());
            e = e.getNextException();
      }

}
```

PreparedStatement

```
PreparedStatement SQL = null;

try {
      SQL = conexion.prepareStatement
                  ( "insert into tabla values (?, ?)" );
      ...

      SQL.setString (1, "Cadena" );
      SQL.setInt    (2,  Entero  );

      if ( SQL.executeUpdate() != -1 )
         throw new Exception ("Error al realizar la actualización");

      ...

} catch (SQLException e) ...
```

CallableStatement

```
CallableStatement cstmt = null;


// Procedimientos

try {
      cstmt = conexion.prepareCall ( "{call proc(?, ?)}" );

      ...

      cstmt.setString (1, "Cadena" );
      cstmt.registerOutParameter (2, java.sql.Types.INTEGER);
      cstmt.executeUpdate();
      System.out.println ( cstmt.getInt(2) );

      ...

} catch (SQLException e) ...


// Funciones

try {
      cstmt = conexion.prepareCall ( "{? = call func(?, ?)}" );

      ...

} catch (SQLException e) ...
```

NOTA: Si el procedimiento no tiene parámetros, podemos utilizar *Statement*. Si el procedimiento tiene parámetros pero no devuelve nada, podemos usar *PreparedStatement*. Si el procedimiento devuelve resultados, tendremos que recurrir a *CallableStatement*.

6. Oracle

Oracle es, sin duda, el servidor de bases de datos relacionales más popular actualmente. Además del conocido ***PL/SQL*** y todas las herramientas de programación de aplicaciones (como SQL/Forms), Oracle destaca por las APIs para distintos lenguajes que ofrece al programador (como ***ODMG*** para C++ o ***JAC*** [*Java Access Classes*] para Java) y los precompiladores existentes para distintos lenguajes (***Pro*C*** [también denominado ***Embedded SQL***], ***Pro*Ada***, ***Pro*COBOL***, ***Pro*FORTRAN***, ***Pro*Pascal*** y ***Pro*PL/I***).

Lo que Oracle ofrece a los programadores de aplicaciones

Lenguaje	APIs	Preprocesadores
Ada		Pro*Ada
C/C++	ODMG (C++)	Pro*C
COBOL		Pro*COBOL
FORTRAN		Pro*FORTRAN
Java	JAC [Java Access Classes]	
Pascal		Pro*Pascal
PL/I		Pro*PL/I

Como cabría esperar de una empresa líder en su sector, Oracle también proporciona controladores para los CLIs ya comentados: ***ODBC*** [*Open DataBase Connectivity*], ***BDE*** [*Borland DataBase Engine*] y ***JDBC*** [*Java DataBase Connectivity*].

6.1 PL/SQL

PL/SQL es un lenguaje de programación estructurado en bloques desarrollado a partir de SQL. Está pensado principalmente para programar triggers y procedimientos almacenados, fragmentos de código que se ejecutan al ocurrir un evento determinado, por lo que es muy limitado en comparación con otros lenguajes de programación de propósito general.

*SQL*Plus*

*SQL*Plus* (pronunciado *"sequel plus"*) es una interfaz en línea de comandos para Oracle. Es un programa que permite introducir, editar, almacenar, recuperar y ejecutar comandos SQL y código PL/SQL.

*Algunos comandos útiles de SQL*Plus*

DESC tabla	Descripción de una tabla
EXIT	Salida de SQL*Plus
HOST orden	Ejecución de una orden del sistema operativo del host
@fichero.sql	Ejecución de las órdenes SQL incluidas en un fichero

Sintaxis de PL/SQL

Comentarios

```
/* Como en C */

-- También como en ADA
```

Operador de asignación

```
:=
```

Estructuras de control

Estructura condicional

```
IF condición
  THEN ...
 [ELSIF condición THEN ...]
 [ELSE ... ]
END IF;
```

Estructuras repetitivas

```
LOOP
 ...
 EXIT WHEN condición
 ...
END LOOP;
```

```
FOR idx IN [REVERSE] min..max LOOP
 ...
END;
```

-- NOTA: La variable índice “idx” no hay que declararla y no existe fuera del bucle.

```
WHILE condición LOOP
 ...
END LOOP;
```

Tipos

TIPOS SIMPLES

BINARY_INTEGER	Entero de 32 bits. Subtipos: NATURAL (0..max) y POSITIVE (1..max)
CHAR(n)	Cadena de longitud fija (n<=32767), por defecto 1 byte. Subtipos: STRING y CHARACTER
DATE	Fecha (*año*+*mes*+*día*+*hora*+*minuto*+*segundo*).
NUMBER(p,s)	Número (precisión p dígitos, escala 10^{-s}). Subtipos: DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER, NUMERIC, REAL y SMALLINT
VARCHAR2(n)	Cadena de longitud variable (n<=32767). Subtipos: VARCHAR

TIPOS COMPUESTOS

```
TYPE tipo IS TABLE OF
  ( CHAR
  | DATE
  | NUMBER
  | variable%TYPE
  | tabla.columna%TYPE )
  [NOT NULL]
INDEX BY BINARY_INTEGER;
```

```
TYPE tipo IS RECORD (
  campo ( tipo
        | variable%TYPE
        | tabla.columna%TYPE
        | tabla%ROWTYPE )
        [NOT NULL]
        [:= expresión] [, ... ]
);
```

Declaración de variables

```
id tipo [:= inicialización];
```

Declaración de constantes

```
id tipo CONSTANT := expresión;
```

Excepciones

Declaración

```
id_excepción EXCEPTION;
```

Elevación

```
RAISE id_excepción;

RAISE; -- Dentro del manejador de la excepción
```

Manejador de excepciones

```
WHEN id_excepción THEN ...

WHEN OTHERS THEN ...
```

Bloques

```
DECLARE
  Parte declarativa
BEGIN
  Cuerpo
EXCEPTION
  Manejo de excepciones
END;
```

Procedimientos y funciones

```
{CREATE OR REPLACE} PROCEDURE id parámetros IS
  Parte declarativa
BEGIN
  Cuerpo
EXCEPTION
  Manejo de excepciones
END id;
```

```
{CREATE OR REPLACE} FUNCTION id parámetros RETURN tipo IS
  Parte declarativa
BEGIN
  Cuerpo
  RETURN expr
EXCEPTION
  Manejo de excepciones
END id;
```

Parámetros

☞ *Parámetros formales*

```
id (IN|IN OUT|OUT) tipo [ := valor_por_defecto ]
```

☞ *Parámetros actuales*

· Como en cualquier otro lenguaje (en orden)

· Identificando los parámetros formales (vg.: uso de valores por defecto):

```
parámetro_formal => expresión
```

Paquetes PL/SQL

Especificación

```
{CREATE OR REPLACE} PACKAGE id IS
  Declaración de tipos, variables, constantes,
  funciones y procedimientos
END id;
```

Cuerpo

```
{CREATE OR REPLACE} PACKAGE BODY id IS
 Declaración de tipos, variables y constantes ocultas al exterior
  Funciones y procedimientos del paquete
BEGIN
  Código de inicialización
END id;
```

Triggers

```
{CREATE OR REPLACE} TRIGGER
 firing specification
BEGIN
 body
END;
```

6.2 Pro*C (Embedded SQL)

Oracle incluye un preprocesador de C que permite incluir sentencias SQL en programas escritos en C: `esql.exe` en Oracle Lite. El precompilador traduce las sentencias SQL a declaraciones y sentencias correctas en C. El código C resultante puede compilarse en cualquier compilador como cualquier otro programa escrito en C.

Las sentencias SQL pueden aparecer en cualquier parte del código, deben estar precedidas por ***EXEC SQL*** y finalizar con punto y coma (**;**). El precompilador traduce lo que haya entre EXEC SQL y el punto y coma final. Además, permite el uso de variables de C (long, short, float, double y char).

EJEMPLO:

```
main()
{
  /* ... */

  EXEC SQL CREATE TABLE T1 (ID INTEGER, NAME CHAR(20), SALARY DOUBLE);

  /* ... */
}
```

Aunque Pro*C permite SQL Dinámico y es bastante flexible, no lo es tanto como ODBC, BDE o JDBC: no permite fijar la consulta u orden SQL en tiempo de ejecución de forma cómoda para el programador. Además, la depuración de los programas que utilizan el preprocesador de Pro*C es bastante tediosa.

7. Estudio comparativo

Para comparar las distintas alternativas expuestas deberíamos tener en cuenta factores de muy diversa índole. En la siguiente tabla se intentan resumir algunas que pueden ser relevantes para el desarrollo de programas de *Data Mining*, aunque quizá carezcan de importancia para realizar aplicaciones típicas de gestión:

Característica	ODBC	BDE	JDBC	PL/SQL	Pro*C
Propietario	MS	Borland	-	Oracle	Oracle
SQL	Sí	Sí	Sí	Sí	Sí
Portabilidad	No	No	Sí	No	No
Estructuras de datos	* * * *	* * * *	* * * *	* *	* * * *
Documentación	* * *	* * * *	* * *	* * * *	* * * *
Facilidad de uso	* *	* * *	* * * *	* * *	* * * *
Facilidad de depuración	* * *	* * *	* * *	* * *	* *
Facilidad de configuración	* * *	* * *	* * *	* * * *	* * *
Facilidad de instalación	* *	* *	* * *	* * * *	* *
Construcción de consultas en tiempo de ejecución	* * * *	* * * *	* * * *	* * *	* * *

☞ *Propietario*: Salvo JDBC, todos los demás estándares son propiedad de alguna casa comercial. Esto puede llevar a los desarrolladores de aplicaciones a una situación de dependencia absoluta respecto a determinadas marcas.

☞ *SQL*: El uso de SQL es el denominador común que comparten todos los métodos descritos de acceso a bases de datos relacionales.

☞ *Portabilidad*: Salvo JDBC, que en teoría debe funcionar en cualquier sistema, cada CLI está pensado para utilizarse únicamente en determinados entornos (vg: ODBC y BDE sólo funcionan bajo Windows, PL/SQL y Pro*C sólo con servidores Oracle).

☞ *Estructuras de datos*: Excepto PL/SQL, los interfaces comentados se utilizan dentro de programas escritos en lenguajes de alto nivel que permiten manejo dinámico de memoria y el uso de estructuras de datos dinámicas. PL/SQL es más limitado en ese sentido (sólo maneja tablas y registros), aunque también puede invocarse desde fuera del servidor Oracle.

☞ *Documentación*: Los APIs suelen estar bien documentados, excepto JDBC (la documentación se limita a enumerar clases y métodos, sin poner ejemplos de uso).

☞ *Facilidad de uso*: JDBC es el interfaz más cómodo de utilizar por parte del programador, lo que no quiere decir que los demás sean difíciles de utilizar.

☞ *Facilidad de depuración*: Salvo Pro*C, todos los demás interfaces son similares a la hora de depurar programas. Pro*C sustituye las órdenes SQL del código por complejas llamadas a rutinas de librerías de Oracle.

☞ *Facilidad de configuración*: PL/SQL no tiene nada que configurar. Pro*C requiere ser adaptado al compilador de C del que dispongamos. Para utilizar JDBC simplemente tenemos que disponer del driver correspondiente (incluirlo en el CLASSPATH) y conocer la especificación del protocolo empleado (URL). ODBC requiere la instalación previa de los controladores que utilicemos, al igual que BDE.

☞ *Facilidad de instalación*: Cuando deseamos instalar un programa en un ordenador diferente al que hemos utilizado en la fase de codificación, ODBC y BDE requieren instalar los controladores utilizados y establecer los alias empleados en el código. Pro*C puede que requiera recompilar el código, JDBC quizá necesite algunos pequeños cambios (establecer las variables de entorno PATH y CLASSPATH y especificar correctamente la URL) y PL/SQL no requiere nada (si Oracle ya está instalado en el ordenador).

☞ *Construcción de consultas en tiempo de ejecución*: Ésta es una característica irrelevante en la mayor parte de las aplicaciones. Sin embargo, para cierto tipo de programas resulta bastante importante. En cierto modo, es como escribir programas que se modifican a sí mismos cambiando en tiempo de ejecución las peticiones SQL que realizan al servidor de bases de datos relacionales. Trabajar con PL/SQL y Pro*C resulta más incómodo que emplear ODBC, BDE o JDBC a la hora de construir consultas con libertad absoluta. En estos últimos, las sentencias SQL no son más que cadenas de caracteres que pueden manipularse libremente. En ese sentido, el manejo de cadenas de caracteres en Java es más cómodo que en los demás lenguajes (C es peor para manipular cadenas y en Delphi coexisten dos tipos de cadenas que sólo complican innecesariamente la labor del programador).

Vistas las cualidades principales de cada una de las técnicas disponibles, podemos descartar inicialmente el uso de Pro*C. Pro*C no introduce nada nuevo respecto a otros estándares que funcionan con distintos servidores de bases de datos y, como se verá en el siguiente apartado, el uso de esos interfaces más versátiles no degrada la eficiencia de los algoritmos utilizados.

Ya que ODBC y BDE son prácticamente idénticos, descartaremos ODBC por razones puramente subjetivas. ODBC no es más que un CLI [*Call-Level Interface*] más (el mejor según Microsoft) y el programador puede elegir según sus gustos. Una mayor familiaridad con las herramientas de desarrollo de Borland puede inclinar la balanza hacia el lado de esta casa comercial. Además, siempre que dispongamos de un driver ODBC podremos emplear BDE (no al contrario).

7.1 Pruebas

"Por otro lado, no podemos olvidarnos de la eficiencia"
John Bentley

Para medir de alguna forma la penalización que supone acceder a una base de datos a través de cada uno de los interfaces descritos, se ha realizado el siguiente experimento:

Se aplica el algoritmo de obtención de los itemsets relevantes utilizado por el ***algoritmo t*** de generación de reglas de asociación (el más simple aplicable a una base de datos relacional) a una pequeña base de datos sintética. El *algoritmo t* no es más que una adaptación directa del *algoritmo Apriori* (propuesto por Agrawal y Skirant en su informe *"Fast Algorithms for Mining Association Rules"*). Este algoritmo se caracteriza porque maneja tablas única y exclusivamente (a pesar de que resulte la forma más adecuada de manejar conjuntos). Por ello es factible su implementación en PL/SQL.

Utilizando una pequeña tabla llamada *TESTCLI* de 5 columnas *(C1, C2, C3, C4, C5)* y 100 tuplas con 10 valores diferentes por columna y fijando el parámetro *MinSupport* en el 3% (3 tuplas), el algoritmo descrito en SQL es el siguiente:

☞ *Generación de L1 (items relevantes)*

```
CREATE TABLE L1 (attr1 INTEGER, val1 INTEGER, support INTEGER);

INSERT INTO L1 SELECT 1,C1, COUNT(*)  FROM TESTCLI GROUP BY C1;
INSERT INTO L1 SELECT 2,C2, COUNT(*)  FROM TESTCLI GROUP BY C2;
INSERT INTO L1 SELECT 3,C3, COUNT(*)  FROM TESTCLI GROUP BY C3;
INSERT INTO L1 SELECT 4,C4, COUNT(*)  FROM TESTCLI GROUP BY C4;
INSERT INTO L1 SELECT 5,C5, COUNT(*)  FROM TESTCLI GROUP BY C5;

DELETE FROM L1 WHERE support < 3;
```

☞ *Generación de C2 (2-itemsets candidatos)*

```
CREATE TABLE C2 ( attr1 INTEGER, val1 INTEGER, attr2 INTEGER, val2 INTEGER);

INSERT INTO C2
SELECT p.attr1,p.val1,q.attr1,q.val1
FROM L1 p, L1 q
WHERE p.attr1<q.attr1;
```

☞ *Obtención de L2 (2-itemsets relevantes)*

```
CREATE TABLE L2
(attr1 INTEGER, val1 INTEGER, attr2 INTEGER, val2 INTEGER, support INTEGER );
```

```
INSERT INTO L2
SELECT p.attr1,p.val1,p.attr2,p.val2,COUNT(*)
FROM C2 p,TESTCLI q
WHERE ((p.ATTR1=1 AND p.VAL1=q.C1)
    OR (p.ATTR1=2 AND p.VAL1=q.C2)
    OR (p.ATTR1=3 AND p.VAL1=q.C3)
    OR (p.ATTR1=4 AND p.VAL1=q.C4)
    OR (p.ATTR1=5 AND p.VAL1=q.C5))
  AND ((p.ATTR2=1 AND p.VAL2=q.C1)
    OR (p.ATTR2=2 AND p.VAL2=q.C2)
    OR (p.ATTR2=3 AND p.VAL2=q.C3)
    OR (p.ATTR2=4 AND p.VAL2=q.C4)
    OR (p.ATTR2=5 AND p.VAL2=q.C5))
GROUP BY p.attr1,p.val1,p.attr2,p.val2;

DELETE FROM L2 WHERE support < 3;
```

☞ *Generación de C3 (3-itemsets candidatos)*

```
CREATE TABLE C3 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER,
                  attr3 INTEGER, val3 INTEGER);

INSERT INTO C3
SELECT p.attr1,p.val1,p.attr2,p.val2,q.attr2,q.val2
FROM L2 p, L2 q
WHERE p.attr1=q.attr1 AND p.val1=q.val1 AND p.attr2<q.attr2;

DELETE FROM C3
WHERE NOT EXISTS ( SELECT * FROM L2
                            WHERE C3.attr2=L2.attr1
                              AND C3.val2=L2.val1
                              AND C3.attr3=L2.attr2
                              AND C3.val3=L2.val2);

DELETE FROM C3
WHERE NOT EXISTS ( SELECT * FROM L2
                            WHERE C3.attr1=L2.attr1
                              AND C3.val1=L2.val1
                              AND C3.attr3=L2.attr2
                              AND C3.val3=L2.val2);

DELETE FROM C3
WHERE NOT EXISTS ( SELECT * FROM L2
                            WHERE C3.attr1=L2.attr1
                              AND C3.val1=L2.val1
                              AND C3.attr2=L2.attr2
                              AND C3.val2=L2.val2);
```

☞ *Obtención de L3 (3-itemsets relevantes)*

```
CREATE TABLE L3 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER,
                  attr3 INTEGER, val3 INTEGER, support INTEGER );
```

```
INSERT INTO L3
SELECT p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3,COUNT(*)
FROM C3 p,TESTCLI q
WHERE ((p.ATTR1=1 AND p.VAL1=q.C1)
    OR (p.ATTR1=2 AND p.VAL1=q.C2)
    OR (p.ATTR1=3 AND p.VAL1=q.C3)
    OR (p.ATTR1=4 AND p.VAL1=q.C4)
    OR (p.ATTR1=5 AND p.VAL1=q.C5))
  AND ((p.ATTR2=1 AND p.VAL2=q.C1)
    OR (p.ATTR2=2 AND p.VAL2=q.C2)
    OR (p.ATTR2=3 AND p.VAL2=q.C3)
    OR (p.ATTR2=4 AND p.VAL2=q.C4)
    OR (p.ATTR2=5 AND p.VAL2=q.C5))
  AND ((p.ATTR3=1 AND p.VAL3=q.C1)
    OR (p.ATTR3=2 AND p.VAL3=q.C2)
    OR (p.ATTR3=3 AND p.VAL3=q.C3)
    OR (p.ATTR3=4 AND p.VAL3=q.C4)
    OR (p.ATTR3=5 AND p.VAL3=q.C5))
GROUP BY p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3;

DELETE FROM L3 WHERE support < 3;
```

☞ *Generación de C4 (4-itemsets candidatos)*

```
CREATE TABLE C4 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER,
                  attr3 INTEGER, val3 INTEGER,
                  attr4 INTEGER, val4 INTEGER);

INSERT INTO C4
SELECT p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3,q.attr3,q.val3
FROM L3 p, L3 q
WHERE p.attr1=q.attr1 AND p.val1=q.val1
  AND p.attr2=q.attr2 AND p.val2=q.val2
  AND p.attr3<q.attr3;

DELETE FROM C4
WHERE NOT EXISTS ( SELECT * FROM L3
                            WHERE C4.attr2=L3.attr1 AND C4.val2=L3.val1
                              AND C4.attr3=L3.attr2 AND C4.val3=L3.val2
                              AND C4.attr4=L3.attr3 AND C4.val4=L3.val3);
DELETE FROM C4
WHERE NOT EXISTS ( SELECT * FROM L3
                            WHERE C4.attr1=L3.attr1 AND C4.val1=L3.val1
                              AND C4.attr3=L3.attr2 AND C4.val3=L3.val2
                              AND C4.attr4=L3.attr3 AND C4.val4=L3.val3);
DELETE FROM C4
WHERE NOT EXISTS ( SELECT * FROM L3
                            WHERE C4.attr1=L3.attr1 AND C4.val1=L3.val1
                              AND C4.attr2=L3.attr2 AND C4.val2=L3.val2
                              AND C4.attr4=L3.attr3 AND C4.val4=L3.val3);
DELETE FROM C4
WHERE NOT EXISTS ( SELECT * FROM L3
                            WHERE C4.attr1=L3.attr1 AND C4.val1=L3.val1
                              AND C4.attr2=L3.attr2 AND C4.val2=L3.val2
                              AND C4.attr3=L3.attr3 AND C4.val3=L3.val3);
```

☞ *Obtención de L4 (4-itemsets relevantes)*

```
CREATE TABLE L4 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER,
                  attr3 INTEGER, val3 INTEGER,
                  attr4 INTEGER, val4 INTEGER, support INTEGER );

INSERT INTO L4
SELECT p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3,p.attr4,p.val4,COUNT(*)
FROM C4 p,TESTCLI q
WHERE ((p.ATTR1=1 AND p.VAL1=q.C1)
    OR (p.ATTR1=2 AND p.VAL1=q.C2)
    OR (p.ATTR1=3 AND p.VAL1=q.C3)
    OR (p.ATTR1=4 AND p.VAL1=q.C4)
    OR (p.ATTR1=5 AND p.VAL1=q.C5))
  AND ((p.ATTR2=1 AND p.VAL2=q.C1)
    OR (p.ATTR2=2 AND p.VAL2=q.C2)
    OR (p.ATTR2=3 AND p.VAL2=q.C3)
    OR (p.ATTR2=4 AND p.VAL2=q.C4)
    OR (p.ATTR2=5 AND p.VAL2=q.C5))
  AND ((p.ATTR3=1 AND p.VAL3=q.C1)
    OR (p.ATTR3=2 AND p.VAL3=q.C2)
    OR (p.ATTR3=3 AND p.VAL3=q.C3)
    OR (p.ATTR3=4 AND p.VAL3=q.C4)
    OR (p.ATTR3=5 AND p.VAL3=q.C5))
  AND ((p.ATTR4=1 AND p.VAL4=q.C1)
    OR (p.ATTR4=2 AND p.VAL4=q.C2)
    OR (p.ATTR4=3 AND p.VAL4=q.C3)
    OR (p.ATTR4=4 AND p.VAL4=q.C4)
    OR (p.ATTR4=5 AND p.VAL4=q.C5))
GROUP BY p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3,p.attr4,p.val4;

DELETE FROM L4 WHERE support < 3;
```

Tras realizar las operaciones descritas, las distintas tablas involucradas en el algoritmo quedan como sigue:

Tabla	*Itemsets*
L1	50
C2	1000
L2	79
C3	21
L3	2
C4	0
L4	0

La prueba se realizó en un Pentium 166MHz con 32MB de memoria RAM y Oracle Lite 3.0 de servidor de bases de datos. Se utilizaron el driver JDBC propio de Oracle, el socket ODBC para el programa que hacía uso de BDE y la versión 3.3 de SQL Plus (para PL/SQL).

☞ *Versión SQL*:

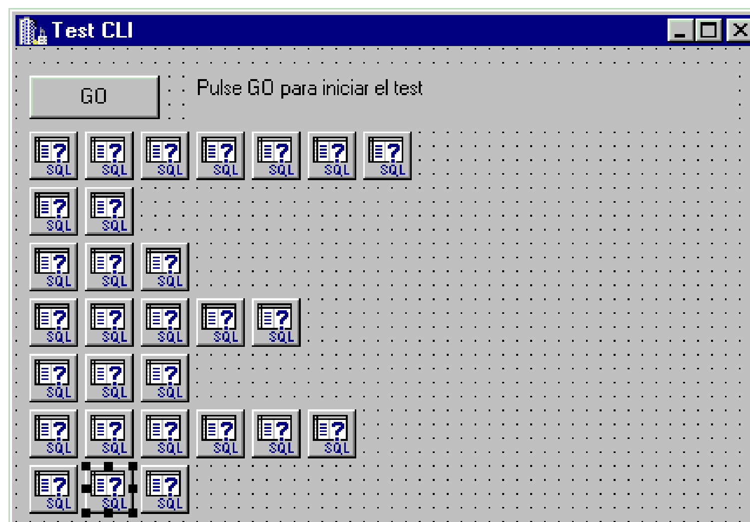
Se ejecutaron directamente las sentencias SQL arriba descritas desde la línea de comandos de SQL Plus. En cada fase del algoritmo se introdujeron los comandos COMMIT necesarios.

☞ *Versión JDBC*:

Se utilizó un programa genérico (aplicable a cualquier tabla) que construía las consultas a partir de la metainformación que él mismo iba obteniendo.

☞ *Versión BDE*:

Se empleó el compilador *C++Builder*. No se utilizó *Delphi*, aunque los resultados habrían sido totalmente equivalente ya que se utilizaron componentes de tipo *TQuery* (comunes a todos los entornos de desarrollo rápido de aplicaciones de Borland):

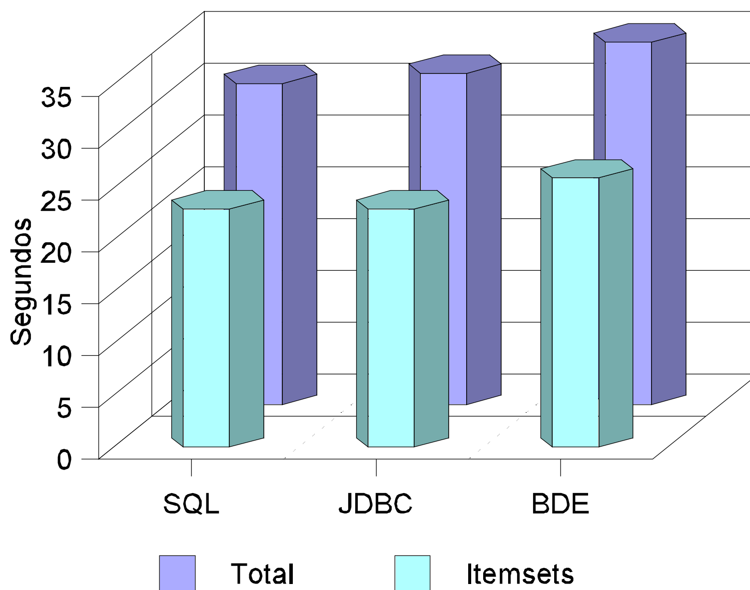


Aspecto del form principal de la aplicación que hace uso de BDE

Los tiempos obtenidos en la prueba fueron los siguientes:

Técnica de acceso empleada (Oracle Lite 3.0)	Tiempo de generación de itemsets	Tiempo total
SQL Plus	23"	31"
JDBC	23"	32"
BDE	26"	35"

Se puede apreciar cómo el uso de JDBC no supone ninguna penalización respecto a la utilización directa de PL/SQL. El mayor tiempo de ejecución requerido a través de BDE puede achacarse al uso de un controlador ODBC desde BDE:



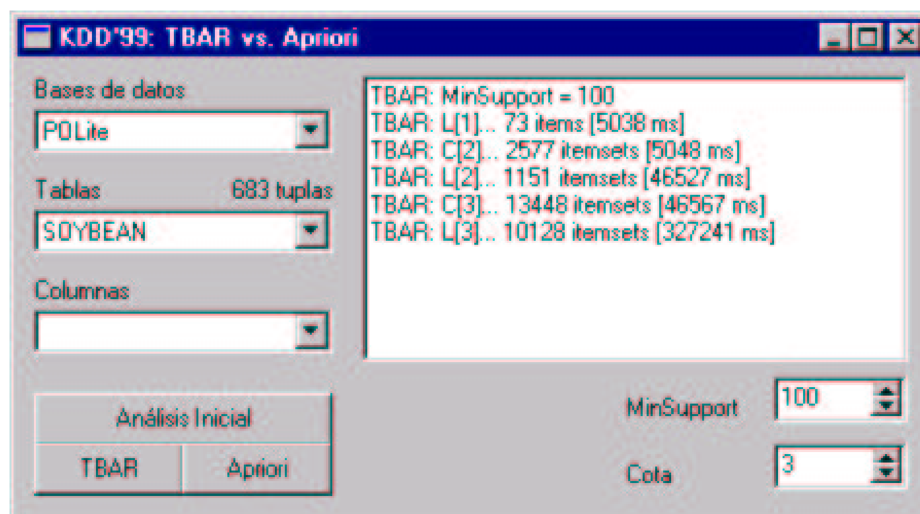
También se han realizado pruebas algo más complejas consistentes en ejecuciones del algoritmo TBAR. Dicho algoritmo se ha implementado en Java (haciendo uso de JDBC) y en C++ (empleando BDE).

Las pruebas se realizaron en un Pentium 166MHz con 32MB de memoria RAM utilizando como un servidor Personal Oracle Lite sobre Windows NT 4 Workstation. Los resultados obtenidos con algunas tablas típicas se resumen a continuación:

Tabla	**Tamaño**	**Algoritmo**	**BDE** ***C++Builder***	**JDBC** ***JDK 1.1***	**JDBC** ***JDK 1.2***
ID3	14x5	TBAR	0.381 s	0.580 s	0.221 s
SOYBEAN	683x36	TBAR	322 s	259 s	69 s
MUSHROOM	8124x23	TBAR	1816 s	688 s	250 s

Como se puede apreciar en la tabla, utilizando JDBC el overhead debido a la comunicación necesaria con el servidor es menor que empleando BDE. Además, esta diferencia se acrecienta conforme el tamaño de la tabla de datos se incrementa.

Por si esto fuese poco, utilizando la versión 1.2 del Java Development Kit de Sun (la cual incluye un compilador JIT [*just-in-time*]), se consiguen mejorar aún más los resultados obtenidos utilizando JDBC, al no tener que ser interpretados los *bytecodes* en tiempo de ejecución.



El aspecto externo del programa realizado con *C++Builder*
que implementa los algoritmos Apriori y TBAR de extracción de reglas de asociación.
Dicho programa accede a la base de datos mediante BDE [Borland Database Engine].

7.2 Conclusiones

"SHYLOCK: No estoy obligado a agradarte con mis respuestas"
El Mercader de Venecia (acto IV, escena 1)
William Shakespeare

Se han descrito brevemente algunas de las opciones de las que dispone un programador a la hora de construir aplicaciones que accedan a bases de datos relacionales. Las alternativas discutidas tienen en común la utilización de SQL a la hora de realizar consultas. De todas ellas, el acceso a través de JDBC ha mostrado ser el más ventajoso de cara al programador:

✐ La disponibilidad de controladores JDBC nativos para distintos servidores de bases de datos garantiza un acceso homogéneo y eficiente a servidores diferentes. Se ha comprobado cómo la eficiencia del acceso a través de JDBC puede llegar a ser comparable con el empleo de herramientas internas de un servidor de bases de datos.

✐ Al ser JDBC parte del API estándar del lenguaje Java, cualquier compilador de Java que cumpla con el estándar debe compilar el código que haga uso del paquete ***java.sql*** (el paquete que incluye las interfaces, clases y excepciones de JDBC).

✐ Al no ser el lenguaje Java propiedad exclusiva de una marca comercial, nos aseguramos de que el esfuerzo realizado en la codificación no depende de los criterios de marketing de una compañía concreta (aunque en Informática todo evoluciona constantemente y no podemos asegurar que dentro de unos años el código escrito para JDBC no quede obsoleto). Al menos, no dependemos de Microsoft (como en ODBC), de Borland (BDE) o de Oracle (Pro*C y PL/SQL).

✐ Al ser Java un versátil lenguaje de alto nivel de propósito general, se pueden implementar aplicaciones que hagan uso de estructuras de datos dinámicas todo lo complejas que se requieran. Esto resulta más incómodo en PL/SQL, que sólo permite manejar tablas.

✐ Java resulta adecuado para manipular cadenas de caracteres con comodidad. Esto permite construir convenientemente consultas en tiempo de ejecución (mejor que en C, donde el manejo de cadenas es más engorroso, o en Delphi, donde coexisten dos tipos de cadenas).

✐ Aunque no sea una razón de peso, hay que reconocer que Java está de moda hoy en día y vende más. Su flexibilidad y presunta portabilidad son las mejores bazas que ofrece este lenguaje de programación orientado a objetos.

✐ Oracle ha anunciado incorporar Java en versiones futuras de sus productos comerciales: Oracle 8i (versión beta, 1998) incorpora una máquina virtual Java. Usar Java interno siempre será más cómodo que emplear PL/SQL. Además, al ser interno, en principio se pueden conseguir buenos tiempos de ejecución para los algoritmos de *Data Mining*.

8. Referencias bibliográficas

SQL

"Developing Applications with Oracle Lite"
Oracle Corporation, 1997.

Fichero PDF (para el Acrobat Reader) incluido con Oracle Lite 3.0 que detalla, en su apéndice B, el catálogo del sistema en Oracle Lite 3.0 (compatible con Oracle 8).

"Oracle7™ Server SQL Reference"
Oracle Corporation, 1996

Completo manual de referencia de SQL para la versión 7 de Oracle.

"Oracle Lite SQL Language Help"
Oracle Corporation, 1997.

Ayuda para Windows de la versión de SQL soportada por OracleLite 3.0.

ODBC

Bill Whiting, Bryan Morgan & Jeff Perkins:
"Teach yourself ODBC Programming in 21 days"
USA: Sams Publishing, 1996

Según los autores de este libro, acérrimos defensores de Microsoft, el estándar ODBC de Microsoft es la mejor alternativa disponible hoy en día para realizar aplicaciones que accedan a bases de datos.

Respecto a los compiladores de Borland, otra de las alternativas disponibles para los programadores bajo Windows, se limitan a comentar que su IDE [*Integrated Development Environment*] "fue un líder de la industria por algún tiempo". Además, según ellos, las bibliotecas OWL de Borland son peores que las MFC de Microsoft (¡cómo no!): no aportan nada nuevo y dejan mucho que desear respecto a las MFC [*Microsoft Foundation Classes*] de Microsoft.

BDE

"Borland Database Engine Online Help"
Borland International, 1997

Fichero de ayuda para Windows incluido con el C++ Builder, el compilador visual de C++ de Borland. En él se describe detalladamente la versión 4.0 de BDE [Borland Database Engine], su API y su utilización.

JDBC

"Oracle Lite JAVA Developer's Guide"
Oracle Corporation, 1997.

En él se comenta (sin demasiados detalles) cómo trabajar con Oracle en Java: procedimientos, triggers, JDBC (Java Database Connectivity) y JAC (Java Access Classes).

"JDK 1.1.1 Documentation"
JavaSoft.

Documentación del JDK (Java Development Kit) que incluye ayuda acerca del paquete **java.sql** que permite acceder a bases de datos programando en Java a través de JDBC.

Oracle

OWENS, Kevin T.
"Building Intelligent Databases with ORACLE PL/SQL, Triggers & Stored Procedures"
USA: Prentice Hall PTR, 1996 [ISBN 0-123-443631-8].

Libro que describe la programación en PL/SQL para Oracle 7.

"Developing Applications with Oracle Lite"
Oracle Corporation, 1997.

Fichero PDF incluido con Oracle Lite 3.0 que describe cómo acceder a bases de datos Oracle desde programas escritos en diversos lenguajes (como Delphi). En su apéndice B, detalla el catálogo del sistema (compatible con Oracle 8).

"Oracle Lite API Reference Guide"
Oracle Corporation, 1997.

Fichero PDF [Portable Document File] destinado para programadores de aplicaciones cliente/servidor con Oracle. Se describen las interfaces de Oracle Lite con C (Embedded SQL C) y C++ (ODMG C++).

"Oracle Lite JAVA Developer's Guide"
Oracle Corporation, 1997.

En él se comenta (sin demasiados detalles) cómo trabajar con Oracle en Java: procedimientos, triggers, JDBC (Java Database Connectivity) y JAC (Java Access Classes).

*"SQL*Plus User's Guide and Reference"*
Oracle Corporation, 1996.

Manual de SQL*Plus 3.3.