



BASES DE DATOS AVANZADAS

Tema 2

Aspectos Avanzados del Modelo Relacional

Univ. Cantabria – Fac. de Ciencias
Francisco Ruiz



Agradecimientos

- Este material ha sido preparado con la colaboración de:
 - Belén Vela (Univ. Rey Juan Carlos)
 - Mario Piattini (Univ. de Castilla-La Mancha)
 - Coral Calero (Univ. de Castilla-La Mancha)



Objetivos

- Conocer el concepto de **comportamiento activo** en una base de datos y aprender a utilizar disparadores (triggers) para incorporar actividad a las bases de datos relacionales.
- Aprender a crear **disparadores en SQL**.
- Conocer las formas de usar el lenguaje **SQL de forma programática** para acceder a los datos desde entornos diferentes al SQL interactivo.
- Aprender a utilizar **SQL embebido**, tanto en forma estática como **dinámica**.
- Aprender la manera en que SQL permite trabajar con **módulos persistentes** (procedimientos y funciones almacenados) y la manera en que se puede acceder a las tablas de la base de datos desde ellos usando **cursores**).



Contenido

- Bases de Datos Activas
 - Comportamiento Activo
 - SGBD Activos
 - Reglas ECA
 - Ejecución
- Disparadores
 - En SQL:2003
- SQL Programático
 - Otras Opciones
- SQL/CLI
 - JDBC
- Módulos Persistentes
 - Rutinas
 - Estructuras de Control
 - Excepciones
- Consultas
- Cursores
- Implementaciones
- SQL Embebido
 - SQLJ
 - Dinámico
 - SQLJ vs JDBC
- Otros Aspectos
 - Recursividad



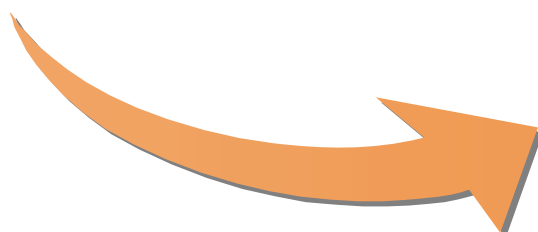
Bibliografía

- **Básica**
 - Piattini et al. (2006): Tecnología y Diseño de Bases de Datos.
 - Cap. 18.
 - Elmasri y Navathe (2007): Fundamentos de Sistemas de Bases de Datos.
 - Caps. 9 y 24.
- **Complementaria**
 - Piattini et al. (2006): Tecnología y Diseño de Bases de Datos.
 - Cap. 8.
 - Connolly y Begg (2005): Sistemas de Bases de Datos.
 - Anexo E.



Bases de Datos Activas

- **Motivación**
 - Nuevas tendencias en BD: GIS, BD Multimedia, BD XML ... **BD Activas**
 - BD convencionales se consideran *muertas* o **pasivas** y no pueden manejar ciertas situaciones
 - **Ejemplo:** actualizar las rutas de un autobús escolar con cada incorporación de nuevos alumnos a la escuela.
 1. Supervisar cada matrícula nueva
 2. Comprobar periódicamente las direcciones de los alumnos matriculados



**Bases de Datos
Activas**



Bases de Datos Activas – Comportamiento Activo

Comportamiento Pasivo



- El marido sabe cómo cocinar
- La esposa solicita **explícitamente** al marido que lo haga
- Roles: objeto fuente (realiza petición: *mujer*) vs objeto receptor (recibe petición: *marido*)



Bases de Datos Activas – Comportamiento Activo

Comportamiento Activo



- El marido sabe cómo cocinar y cuándo cocinar
- Roles: objeto observador (*marido*) VS objeto observado (*mujer*)



Bases de Datos Activas – Comportamiento Activo

Comportamiento Activo

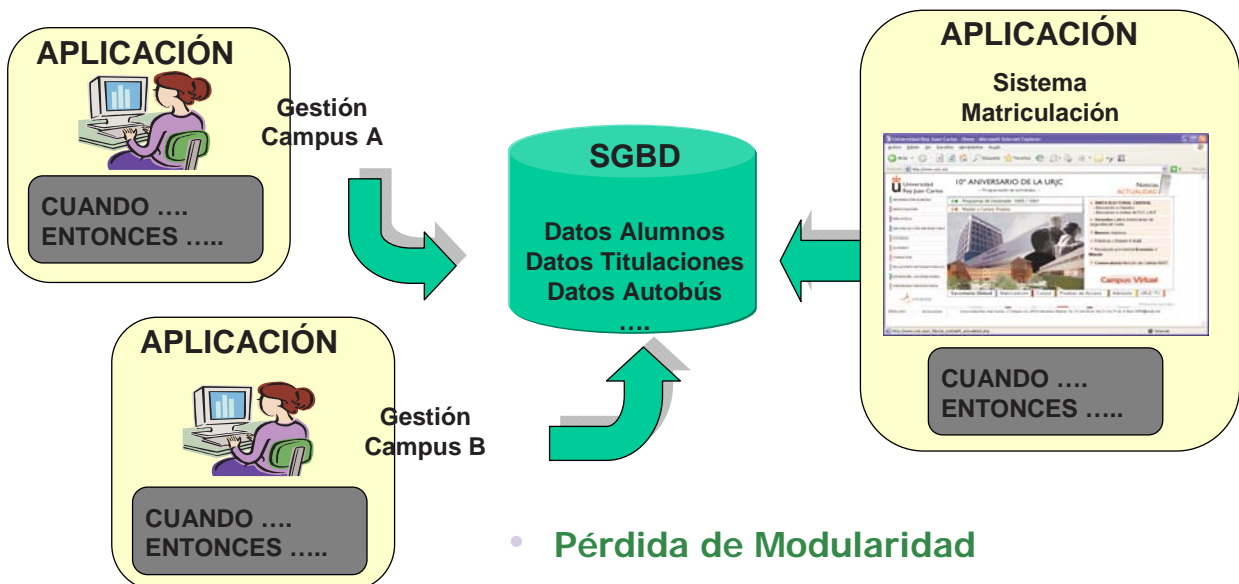
- Comportamiento Activo = CUÁNDO + QUÉ
- Ejemplos:
 - Gestión de Stocks:
cuando ITEM < 10
entonces **solicitar nuevo ITEM** al proveedor
 - Productos Perecederos
cuando producto.caduca - fecha_actual < 7
entonces **reducir el precio del producto**
 - Gestión de Autobuses
cuando autobús lleno y falte más de una semana para el viaje
entonces **poner autobús adicional**



Bases de Datos Activas – Comportamiento Activo

Comportamiento Activo

- Podemos recoger ese comportamiento en las aplicaciones que acceden a la BD



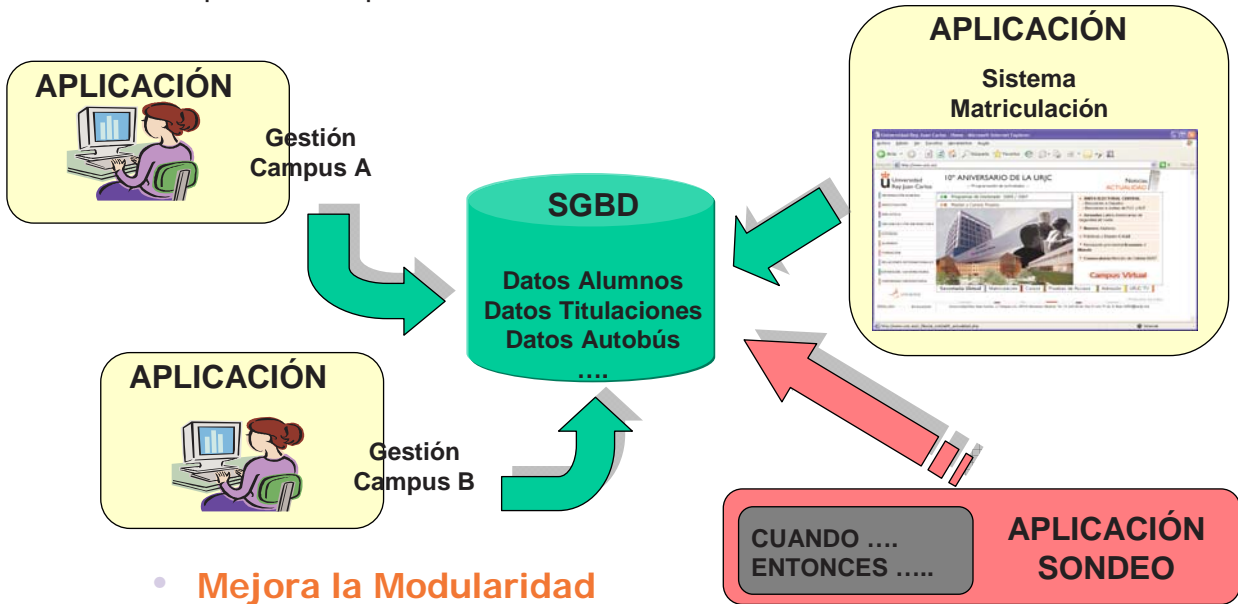
- Pérdida de Modularidad
- Semántica Distribuida



Bases de Datos Activas – Comportamiento Activo

Comportamiento Activo

- O en una aplicación específica.



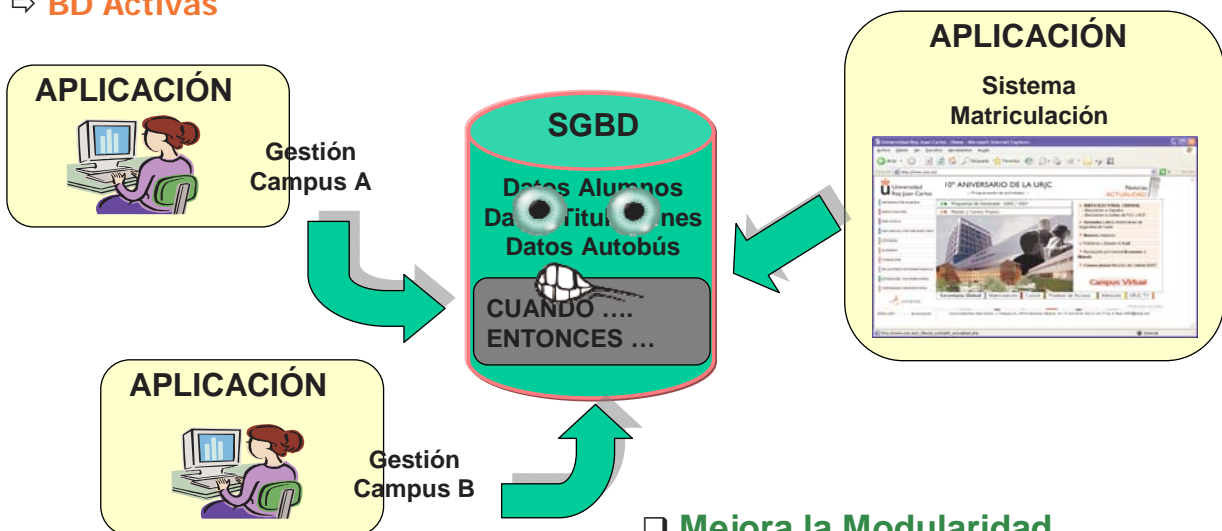
- **Mejora la Modularidad**
- **Frecuencia de Sondeo??**



Bases de Datos Activas – Comportamiento Activo

Comportamiento Activo

- Podemos recoger ese comportamiento en la propia BD
- ⇒ **BD Activas**



- ❑ **Mejora la Modularidad**
- ❑ **Reacción Inmediata**



Bases de Datos Activas – Comportamiento Activo

- **Manifiesto de las Bases de Datos Activas (1996)**
 - Características de un **SGBD Activo**:
 - Tiene un modelo de reglas ECA.
 - Soporta la gestión de reglas y la evolución de la base de datos.
 - Características de **ejecución de reglas ECA**:
 - EL SGBD tiene un modelo de ejecución.
 - Ofrece diferentes modelos de acoplamiento.
 - Implementa modos de consumo.
 - Gestiona la historia de eventos.
 - Implementa la resolución de conflictos.
 - Características de **aplicación y usabilidad**:
 - Posee un entorno de programación.
 - Es ajustable.



Bases de Datos Activas – SGBD Activos

SGBD Activo

- Los SGBD Activos proporcionan mecanismos para:
 - Definir el **qué** y el **cuándo**
⇒ **Modelo de Conocimiento**
 - Realizar un seguimiento del **cuándo** y gestionar el **qué**
⇒ **Modelo de Ejecución**





Bases de Datos Activas – SGBD Activos

- **Modelo de Conocimiento**
 - Describe la situación y la reacción correspondiente (**reglas ECA**).
- **Modelo de Ejecución**
 - Realiza un seguimiento de la situación y gestiona el comportamiento activo (cómo se comportan las reglas en tiempo de ejecución).



Bases de Datos Activas – SGBD Activos

SQL y las BD Activas

- **CONSTRAINTS**: especificaciones del DDL que se aplican a columnas o tablas.
 - Conviene darles nombre
 - UNIQUE, NOT NULL, REFERENCES, CHECK

```
ALTER TABLE Dept_tab  
ADD PRIMARY KEY (Deptno);
```

```
ALTER TABLE Emp_tab  
ADD FOREIGN KEY (Deptno)  
REFERENCES Dept_tab(Deptno);
```

- **ASERCIONES** (*ASSERTIONS*): restricción que no tiene por qué estar asociada a una única tabla.
- **TRIGGERS** (*DISPARADORES*): aserciones con acciones asociadas.



Aplicaciones de los SGBD Activos

- **Internas:** clásicas de la utilización o administración de BD
 - Control de integridad (*ON UPDATE CASCADE*)
 - Mantenimiento de datos derivados (vistas)
 - Administración de copias (monitorizar y registrar cambios)
 - Seguridad y auditoría
 - Gestión de versiones
- **Externas:** *reglas de negocio* (p.e. *Rutas de Autobús*)
- Ventajas:
 - Mayor productividad
 - Mejor mantenimiento
 - Reutilización de código
 - Reducción del tráfico de mensajes



- **Aplicaciones**
 - **Notificación** cuando ocurren ciertas condiciones
 - **Reforzar** las restricciones de **integridad**
 - Los disparadores son más inteligentes y más potentes que las restricciones.
 - **Mantenimiento de datos derivados**
 - Actualización automática de datos derivados evitando anomalías debidas a la redundancia.
 - Ejemplo: un disparador actualiza el saldo total de un cuenta bancaria cada vez que se inserta, elimina o modifica un movimiento en dicha cuenta.



Bases de Datos Activas – Reglas ECA

Modelo del Conocimiento: Definir QUÉ y CUÁNDO

- Reglas **ECA** ⇨ **Evento** – **Condición** – **Acción**
 - **Evento**: qué dispara la acción
 - **Condición**: estado que debe darse
 - **Acción**: qué se hace



Bases de Datos Activas – Reglas ECA

• Evento

- **FUENTE**: ¿Qué ocasiona la ocurrencia de un evento?
 - una instrucción del LMD (antes o después):
insert, delete, update, select
 - una instrucción para la gestión de transacciones: *commit, abort*
 - una excepción: violación de autorizaciones, bloqueos, etc.
 - el reloj: el 28 de Mayo a las 19:30h.
 - la aplicación (externo a la BD).
- **GRANULARIDAD**: ¿Qué cambios considera UNA ocurrencia del evento?
 - cambios en sólo una tupla (*disparadores a nivel de fila*):
1 tupla : 1 evento.
 - cambios en todas las tuplas (*disparadores a nivel de sentencia*):
0..n tuplas : 1 evento.



Bases de Datos Activas – Reglas ECA

Condición y Acción

- **CONDICIÓN:**
 - Un predicado sobre la BD: consulta
 - Puede ser opcional (si no se incluye se considera que la condición es siempre cierta)
- **ACCIÓN:** ¿Qué se puede incluir en la reacción?
 - Operación en la BD (órdenes de SQL, insert, delete, ...)
 - Comandos de SQL extendido (p.e. PL/SQL)
 - Llamadas externas (envío de mensajes)
 - Abortar la transacción
 - Hacer en lugar de (*instead-of*)



Bases de Datos Activas – Reglas ECA

Restricción: “*Ningún empleado debe ganar más que su jefe*”



Temporalidad

after

insert or update on Empleado

Evento

Condición

```

if new.sueldo > (select B.sueldo
from Empleado B
where B.nombre = new.nombreJefe)

```

Acción

```

do update Empleado
  set sueldo = new.sueldo
  where nombre = new.nombreJefe

```



Bases de Datos Activas – Ejecución

Modelo de Ejecución



Francisco Ruiz - BDA

2.23



Bases de Datos Activas – Ejecución

- En general, la forma en que se ejecutan dichas fases depende de dos **modos de acoplamiento**:
 - Evento vs Condición
 - Condición vs Acción
- Para ambos modos las **opciones** son:
 - **Inmediato** (*immediate*)
 - La condición se evalúa inmediatamente después del evento.
 - La acción se ejecuta inmediatamente después de la condición.
 - **Diferido** (*deferred*)
 - La condición se evalúa al final de la transacción.
 - La acción se ejecuta al final de la transacción.
 - **Desprendido** (*detached*)
 - La condición se evalúa en una transacción diferente.
 - La acción se ejecuta en una transacción diferente.

Francisco Ruiz - BDA

2.24



Bases de Datos Activas – Ejecución

Modos de Acoplamiento

¿Cuándo se evalúa la condición?

“Ningún empleado debe ganar más que su jefe”

NOMBRE	TIPO	SUELDO
Juan	Empleado	1000
Sara	Directiva	1050

← Jefa de Juan

UPDATE EMPLEADO
SET SUELDO = SUELDO * 1.10

1º

NOMBRE	TIPO	SUELDO
Juan	Empleado	1100
Sara	Directiva	1050

1º

NOMBRE	TIPO	SUELDO
Juan	Empleado	1000
Sara	Directiva	1155

2º

NOMBRE	TIPO	SUELDO
Juan	Empleado	1100
Sara	Directiva	1100

2º

NOMBRE	TIPO	SUELDO
Juan	Empleado	1100
Sara	Directiva	1155

3º

NOMBRE	TIPO	SUELDO
Juan	Empleado	1100
Sara	Directiva	1200

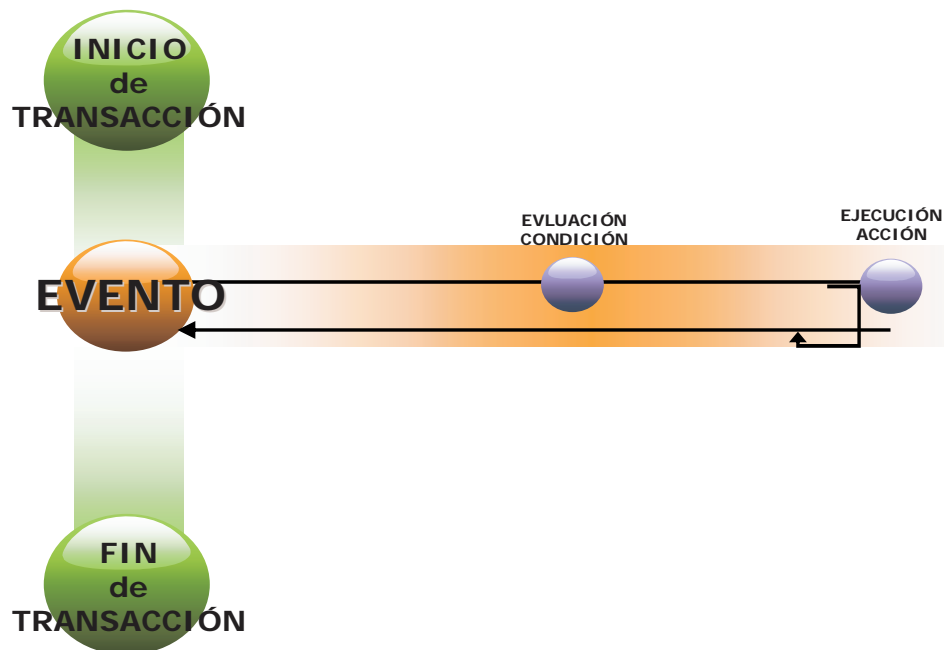
Solución:
Diferir comprobación de restricciones 2.25



Bases de Datos Activas – Ejecución

Modos de Acoplamiento

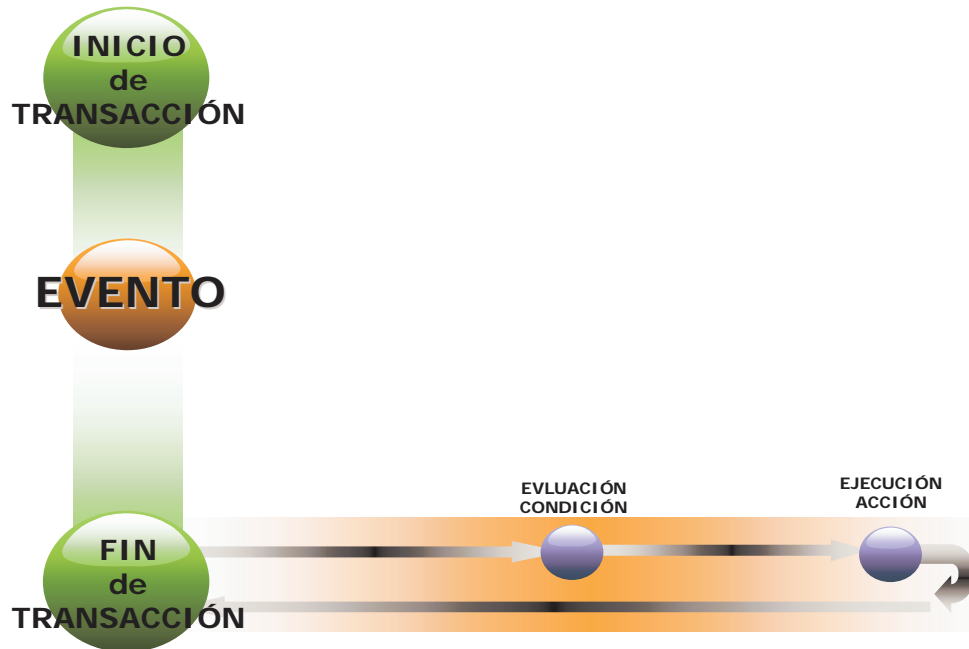
- Modelo de Acoplamiento **Inmediato**





Modos de Acoplamiento

- Modelo de Acoplamiento **Diferido**



- **Disparador** (trigger)
 - Está asociado a una única tabla base.
 - Es el concepto clave para implementar **BD activas**.
 - Tiene tres partes:
 - Un **evento**: indica la acción sobre la tabla base que causará que se active el disparador.
 - **INSERT**, **DELETE**, o **UPDATE**
 - Un **tiempo de acción**: indica cuando se activará el disparo.
 - **BEFORE** => antes del evento.
 - **AFTER** => después del evento.
 - Una **acción**: Se llevan a cabo si ocurre el evento. Puede ser de dos tipos:
 - Una sentencia SQL ejecutable (SQL executable statement).
 - Un bloque atómico de sentencias SQL ejecutables.



Disparadores – En SQL:2003

```
<definición de disparador> ::=
  CREATE TRIGGER <nombre del disparador> { BEFORE | AFTER } <evento>
  ON <nombre tabla> [ REFERENCING <lista de tablas/variables de transición> ]
  <acción disparada>

<evento> ::=
  { INSERT | DELETE | UPDATE [ OF <lista de columnas> ] }

<lista de tablas/variables de transición> ::=
  { OLD [ ROW ] [ AS ] <nombre de variable>
  | NEW [ ROW ] [ AS ] <nombre de variable>
  | OLD TABLE [ AS ] <nombre de tabla>
  | NEW TABLE [ AS ] <nombre de tabla> } ...

<acción disparada> ::=
  [ FOR EACH { ROW | STATEMENT } ]
  [ WHEN (<condición de búsqueda>) ]
  { <sentencia SQL ejecutable>
  | BEGIN ATOMIC { <sentencia SQL ejecutable> ; } ... END
```



Disparadores – En SQL:2003

Sean las dos tablas siguientes:

DEPT (N_DEPT, LOCALIDAD, ..., NUM_EMP)

EMP (COD_EMP, NOMBRE, N_DEPT)

Clave ajena EMPLEADO.N_DEPT→DEPT

```
CREATE TRIGGER borrar_emp
  AFTER DELETE ON emp
  FOR EACH ROW
  (UPDATE dept.num_emp = dept.num_emp -1
   WHEN dept.n_dept = emp.n_dept);
CREATE TRIGGER insertar_emp
  AFTER INSERT ON emp
  FOR EACH ROW
  (UPDATE dept.num_emp = dept.num_emp + 1
   WHER dept.n_dept = emp.n_dept);
```



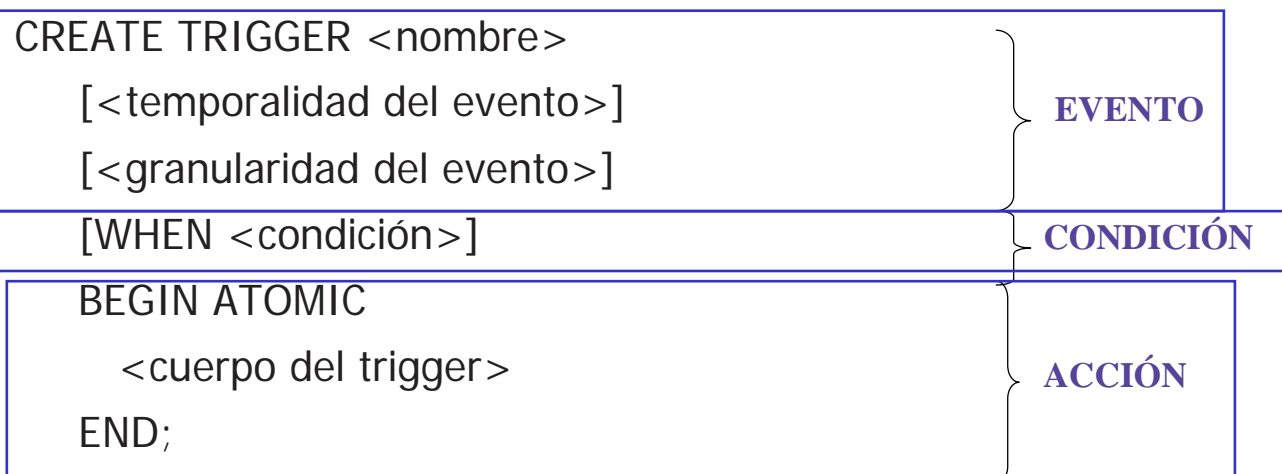
Disparadores – En SQL:2003

```
CREATE TRIGGER modificar_emp
  AFTER UPDATE OF n_dept ON emp
  REFERENCING OLD ROW AS v_emp
  NEW ROW AS n_emp
  FOR EACH ROW
  (UPDATE dept
   SET num_emp = num_emp + 1
   WHERE dept.n_dept = n_emp.n_dept);
(UPDATE DEPT
 SET num_emp = num_emp -1
 WHERE Dept.n_dept = v_emp.n_dept);
```



Disparadores – En SQL:2003

Estructura de la orden para crear un disparador





Disparadores – En SQL:2003

{ BEFORE | AFTER } <evento> ON <nombre tabla>

- **Temporalidad del evento**
 - **BEFORE** Operación → El cuerpo del disparador debe ejecutarse antes del evento que causa la activación del disparador
 - **AFTER** Operación → El cuerpo del disparador debe ejecutarse después del evento que causa la activación del disparador

Evento: INSERT, DELETE O UPDATE

Ej. AFTER DELETE ON Alumnos;
BEFORE UPDATE OF credits ON Asignaturas;



Disparadores – En SQL:2003

[FOR EACH { ROW | STATEMENT }]

- **Granularidad del evento**
 - **FOR EACH ROW**
 - El disparador es a nivel de fila.
 - El cuerpo del disparador se debe ejecutar fila a fila a la tabla afectada.
 - **FOR EACH STATEMENT**
 - Es la opción por defecto.
 - El disparador es a nivel de orden.
 - El cuerpo del disparador se debe aplicar a toda la tabla a la vez.



Disparadores – En SQL:2003

[**WHEN** (<condición de búsqueda>)]

- Sólo se define para disparadores a nivel de fila.
 - Operadores relacionales:
< <= > >= = <>
 - Operadores lógicos:
AND, OR, NOT



Disparadores – En SQL:2003

<lista de tablas/variables de transición> ::=
{ **OLD** [**ROW**] [**AS**] <nombre de variable>
| **NEW** [**ROW**] [**AS**] <nombre de variable>
| **OLD TABLE** [**AS**] <nombre de tabla>
| **NEW TABLE** [**AS**] <nombre de tabla> } ...

- Sirven para referir a los **valores antes y después** de la acción.
- En el cuerpo del disparador se referencian como :OLD ó :NEW.



Disparadores – En SQL:2003

- Con **OLD**. <nombre_columna> referenciamos:
 - al valor que tenía la columna antes del cambio debido a una modificación (UPDATE)
 - al valor de una columna antes de una operación de borrado sobre la misma (DELETE)
 - al valor NULL para operaciones de inserción (INSERT)
- Con **NEW**. <nombre_columna> referenciamos:
 - Al valor de una nueva columna después de una operación de inserción (INSERT)
 - Al valor de una columna después de modificarla mediante una sentencia de modificación (UPDATE)
 - Al valor NULL para una operación de borrado (DELETE)



Disparadores – En SQL:2003

```
{ <sentencia SQL ejecutable>  
| BEGIN ATOMIC { <sentencia SQL ejecutable> ; } ...  
END
```

- En el **cuerpo del disparador** se pueden incluir:
 - Una sentencia SQL ejecutable
 - Un bloques de sentencias.



Disparadores – En SQL:2003

- **Ejemplo** – Granularidad de Fila

```
CREATE TRIGGER Ejemplo_fila
  AFTER DELETE ON tabla1
  REFERENCING OLD ROW AS v
  FOR EACH ROW
  WHEN ((v.nombre='pepe') OR (v.edad > 35))
  BEGIN ATOMIC
    DELETE FROM tabla2 WHERE tabla2.cod=v.cod;
  END;
```



Disparadores – En SQL:2003

- **Ejemplo** – Granularidad de Sentencia (Tabla)

```
CREATE TRIGGER Ejemplo_sentencia
  AFTER DELETE ON tabla1
  REFERENCING OLD AS anterior
  BEGIN ATOMIC
    DELETE FROM tabla2 WHERE
      tabla2.cod=anterior.cod;
  END;
```



SQL Programático

- **Objetivo:**
 - Acceder a una base de datos desde un programa de aplicación en vez de desde una interfaz SQL interactiva.
- **¿Por qué?**
 - Una interfaz SQL interactiva es conveniente pero no es suficiente.
 - La mayoría de las operaciones sobre bases de datos se realizan a través de programas de aplicación (incluidas aplicaciones web).



SQL Programático

- Hay varias maneras diferentes de hacer lo anterior (**SQL programático**):
 - **Biblioteca de Funciones de Base de Datos**
 - Conjunto de llamadas a base de datos disponibles en un lenguaje host.
 - Tipo especial de APIs (*Application Program Interface*).
 - **Código almacenado en la propia base de datos**
 - En un lenguaje especial de forma que las **incompatibilidades** con el modelo de datos se minimizan.
 - **SQL Embebido**
 - Órdenes de base de datos (normalmente SQL) están incrustadas en el código de un lenguaje de programación de propósito general (Java, C, ...).



SQL Programático

- **SQL:2003** establece los mismos tres **métodos de comunicación** (*binding styles*) entre clientes y servidores SQL:
 - **CLI** (Call-Level Interface) *[parte 3 del estándar]*
 - **Módulos Persistentes** *[parte 4 del estándar]*
 - **SQL Embebido** *[parte 10 para Java]*
[parte 2 para otros lenguajes]
- Aparte, está la opción de **invocación directa** *[parte 2 del estándar]*
 - Ejecutar órdenes SQL de forma directa a través de un interfaz interactivo entre el usuario y el servidor SQL.



SQL Programático

- En SQL/CLI y SQL Embebido la manera habitual de proceder es:
 1. El **programa cliente abre una conexión** con el **servidor de base de datos**.
 2. El programa cliente envía **consultas y/o actualizaciones** a la base de datos a través de dicha conexión y recibe los resultados.
 3. Cuando ya no es necesario volver a acceder a la base de datos, el programa cliente **cierra la conexión**.



SQL Programático

- **Incompatibilidades** entre el lenguaje de programación host y el modelo de datos.
 - En **Tipos**
 - Establecer equivalencias y conversiones entre tipos.
 - **Desajuste de Impedancia** con el modelo de datos
 - Mecanismo de uso de conjuntos de filas vs un registro cada vez.
 - Necesidad de iteradores especiales para recorrer los resultados de consultas y manipular valores individuales.



SQL Programático – Otras Opciones

- **SQL:2003** desarrolla otros aspectos programáticos en varias partes adicionales del **estándar**:
 - Parte 9: **MED (Management of External Data)**
 - Acceder a datos externos, fuera del control de un servidor SQL.
 - Parte 13: **JRT (Java Routines and Types)**
 - Usar métodos y clases Java desde SQL como si fueran rutinas SQL y tipos estructurados SQL respectivamente.
 - Parte 14: **XRS (XML-Related Specifications)**
 - Uso de XML desde SQL.



SQL Programático – Otras Opciones

- Combinaciones posibles al invocar rutinas desde o hacia SQL.

Invocadas desde	Escritas en	
	SQL	Otro lenguaje
SQL	<i>Funciones y Procedimientos SQL (PSM)</i>	<i>Funciones y Procedimientos externos (JRT)</i>
Otro lenguaje	<i>Procedimientos invocados externamente (Embebido)</i>	



SQL Programático – Otras Opciones

- **SQL/MED:**
 - Extiende SQL para soportar la gestión de datos externos a través del uso de wrappers (*envoltorios*) de datos externos y tipos "*datalink*".
 - **Wrapper:**
 - Colección de rutinas invocables desde un servidor SQL, que dan soporte para el acceso a datos externos.
 - Provee acceso virtual a datos como si fueran una "tabla externa".
 - Cada wrapper da acceso a uno o varios servidores externos.
 - **Datalink:**
 - Valor que referencia un fichero que no es parte de un entorno SQL y que es gestionado externamente al servidor SQL.
 - Archivos TXT, XLS, ...



SQL Programático – Otras Opciones

- **SQL/MED:**

- Un **descriptor** de un “**Foreign-Data Wrapper**” es un elemento de un catálogo SQL que incluye:
 - Nombre
 - Identificador de autorización del propietario del descriptor.
 - Nombre del lenguaje en el cual está escrito el wrapper.
 - Opciones varias (sin especificar)...

```
CREATE FOREIGN DATA WRAPPER <nombre>  
  [ LIBRARY <nombre biblioteca> ] <lenguaje usado>  
  [ <opciones genéricas> ]
```



SQL Programático – Otras Opciones

- **SQL/MED:**

- Un **Datalink** está representado conceptualmente por:
 - **Referencia al fichero:** cadena que apunta al fichero externo.
 - **Indicador lectura mediatizada** (*SQL-Mediated Read Access Indication*): Si/No el fichero sólo puede ser leído a través de las operaciones especialmente provistas para ello.
 - **Indicador escritura mediatizada** (*SQL-Mediated Write Access Indication*): Si/No el fichero sólo puede ser modificado a través de las operaciones especialmente provistas para ello.
 - **Elemento de Escritura** (Write Token): Un valor que representa un elemento usado para leer o modificar el fichero.
 - **Indicación de Construcción:** Cadena que indica la manera en que se construyó (NEWCOPY, PREVIOUSCOPY, ..).



SQL/CLI

- **SQL/CLI:**
 - Especifica un método para enlazar con un servidor SQL desde un programa de aplicación, escrito en un lenguaje de programación estándar.
 - Define las estructuras y procedimientos para ejecutar sentencias SQL desde dentro de un programa de forma que los procedimientos empleados son independientes de las sentencias SQL ejecutadas.
 - El efecto es funcionalmente equivalente al SQL Dinámico (SQL/OLB).
 - **ODBC** (Open Database Connectivity) es una alternativa industrial de Microsoft.



SQL/CLI

- **SQL/CLI:**
 - Los **procedimientos** pueden ser usados para:
 - Localizar y liberar recursos (área descriptor y área de comunicación).
 - Inicializar, controlar y terminar conexiones entre el cliente SQL y el servidor.
 - Causar la ejecución de sentencias SQL, incluyendo preparar sentencias para ejecución subsecuente.
 - Obtener información de diagnóstico.
 - Obtener información sobre los servidores a los que el cliente se puede conectar.



SQL/CLI

- **SQL/CLI** - Diferencias con el **SQL Embebido**:
 - Ciertas librerías (sqlcli.h en C) tienen que ser instaladas y estar disponibles).
 - A cambio, no requiere precompilación.
 - Sentencias SQL se crean de forma dinámica y son pasadas como parámetros de tipo string en las llamadas.



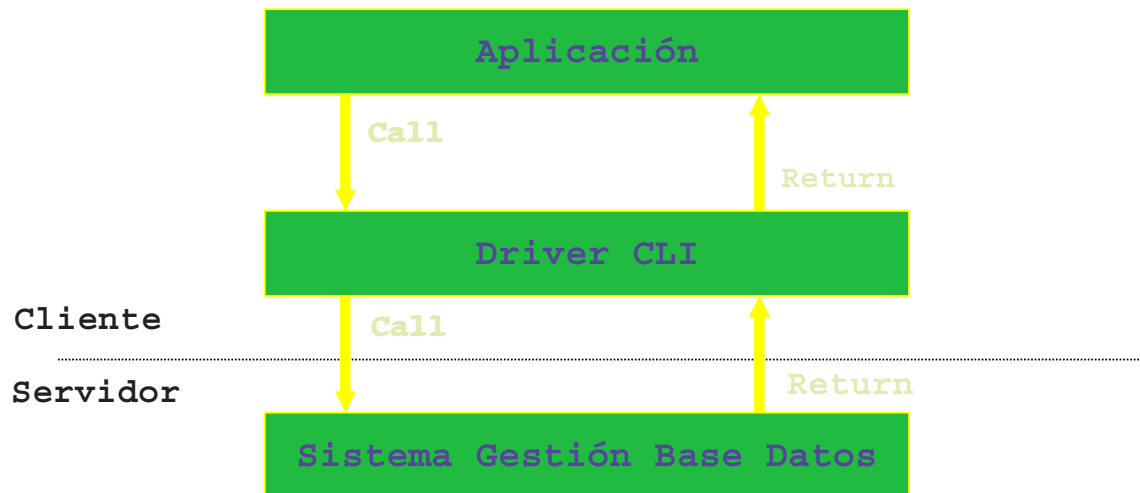
SQL/CLI

- **SQL/CLI**:
 - Se manejan varias **estructuras** (structs en C) para guardar datos de conexión y trabajo con la base de datos:
 - De **Entorno**: instalación del servidor SQL.
 - De **Conexión**: información necesaria para cada conexión particular.
 - De **Sentencia**: información necesaria para pasar una sentencia SQL a una conexión.
 - De **Descripción**: filas de datos resultantes de una consulta, o parámetros de una sentencia.



SQL/CLI

- **SQL/CLI:**



SQL/CLI

- Pasos a realizar en lenguaje C:

1. Cargar librerías SQL/CLI
2. Declarar variables de manejo de registros (usando **SQLHSTMT**, **SQLHDBC**, **SQLHENV**, **SQLHDEC**)
3. Preparar un registro de entorno usando **SQLAllocHandle**
4. Preparar un registro de conexión usando **SQLAllocHandle**
5. Preparar un registro de sentencia usando **SQLAllocHandle**
6. Preparar una sentencia usando la función **SQLPrepare**
7. Asociar parámetros a las variables de programa
8. Ejecutar la sentencia SQL invocando **SQLExecute**
9. Asociar columnas de la consulta a variables C usando **SQLBindCol**
10. Usar **SQLFetch** para recuperar valores de columnas en variables C



- **SQL/CLI – Funciones:**
 - **SQLAllocHandle(T,I,O)** se emplea para crear las estructuras o manejadores antes comentados.
 - Parámetros:
 - T = tipo (ej., SQL_HANDLE_STMT).
 - I = manejador de entrada (estructura en el siguiente nivel superior: sentencia < conexión < entorno).
 - O = manejador de salida.
 - **SQLPrepare(H,S,L).**
 - Hace que el string S , de longitud L , sea interpretado como una sentencia SQL y optimizado. La sentencia ejecutable resultante es situada en el manejador de sentencia H .



- **SQL/CLI – Funciones:**
 - **SQLExecute(H).**
 - La sentencia SQL representada por el manejador de sentencia H es ejecutada.
 - **SQLExecuteDirect(H,S,L).**
 - Combina las anteriores SQLPrepare y SQLExecute cuando la sentencia S solo se ejecutará una única vez.
 - Como antes, H es el manjeador de sentencia y L es la longitud del string S .



- **SQL/CLI – Funciones:**

- **SQLFetch(H).**

- Devuelve la siguiente fila (la primera la primera vez) del resultado de la consulta indicada en la sentencia con manejador H.
- Un cursor es declarado de forma implícita al ejecutar una sentencia.

- **SQLBindCol(H, CN, BT, D, BL, L).**

- Para la fila obtenida con el último SQLFetch de la consulta asociada al manejador H, asocia la columna nº CN a la variable de dirección D, con una longitud de caracteres (si procede) de L.
- BT = tipo de buffer, BL = longitud del buffer.



- **SQL/CLI – Ejemplos:**

```
SQLAllocHandle(SQL_HANDLE_STMT, myCon,  
myStat);
```

siendo

- `myCon` un manejador de conexión previamente creado.
- `myStat` nombre del manejador de sentencia que será creado.



- **SQL/CLI – Ejemplos:**

```
SQLPrepare(myStat, "SELECT cerveza, precio  
FROM Ventas WHERE bar = 'Gambrinus'",  
SQL_NTS);  
SQLExecute(myStat);
```

Esta constante indica que el segundo argumento es un "null-terminated string"; es decir, no se dispone de una cifra exacta de caracteres.

```
SQLBindCol(myStat, 1, , &cerveza, , );  
SQLBindCol(myStat, 2, , &precio, , );
```



- **SQL/CLI – Ejemplos:**

```
while ( SQLFETCH(myStat) != SQL_NO_DATA)  
{  
    /* hacer algo con la cerveza y el  
    precio */  
}
```

*Macro CLI representando el valor de estado
SQLSTATE = 02000 => no se ha encontrado
ninguna fila de resultados.*



SQL/CLI - JDBC

- **Java Database Connectivity (JDBC)** es una biblioteca de funciones similar a SQL/CLI, pero con Java como lenguaje base.
 - No es un estándar oficial.
- Existen algunas diferencias entre ambas.



SQL/CLI - JDBC

- Haciendo una **conexión**.

```
import java.sql.*;
Class.forName("com.mysql.jdbc.Driver");
Connection myCon =
    DriverManager.getConnection(...);
```

clases JDBC

aquí van la dirección y nombre de la base de datos, el usuario y la password.

driver para MySQL (hay otros)



SQL/CLI - JDBC

- JDBC provee dos clases para el **manejo de sentencias SQL**:
 - *Statement* = un objeto que puede aceptar un string conteniendo una sentencia SQL y puede ejecutar dicha sentencia.
 - *PreparedStatement* = un object que tiene una sentencia SQL asociada lista para ser ejecutada.
- La clase *Connection* tiene métodos para crear objetos de ambas clases.



SQL/CLI - JDBC

- Creando **sentencias**.

```
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 =
myCon.createStatement(
    "SELECT beer, price FROM Sells " +
    "WHERE bar = 'Joe' 's Bar' "
);
```

*sin argumentos retorno una Statement;
con un argumento retorna una
PreparedStatement.*



SQL/CLI - JDBC

- Ejecutando **consultas y modificaciones**.
 - JDBC distingue entre consultas (queries) y modificaciones (updates).
 - Statement y PreparedStatement tienen métodos para ejecutar ambas:
 - `executeQuery`
 - `executeUpdate`
 - Para Statements se debe pasar la consulta o modificación como argumento.
 - Para PreparedStatement no hay argumentos (ya se dió la sentencia antes).



SQL/CLI - JDBC

- **Ejemplos** de consultas y modificaciones.

```
stat1.executeUpdate(  
    "INSERT INTO Ventas VALUES ('Heineken', 40)"  
);
```

```
ResultSet menu = prestat2.executeQuery();
```

Una PreparedStatement conteniendo una consulta.



- Accediendo a los **resultados**.
 - **ExecuteQuery** retorna un objeto de la clase **ResultSet**, que es parecido a un cursor.
 - El método **next()** avanza el "cursor" a la siguiente fila de resultados.
 - La primera vez que se aplica se obtiene la primera fila.
 - Si no hay más filas, next() retorna el valor **false**.
 - El método **getX(i)**, donde X indica el nombre de un tipo de dato (ej. Integer), devuelve el valor de la columna i-ésima.
 - El tipo de dicha columna debe ser el mismo X.



- **Ejemplo** de acceso a resultados.

```
while ( menu.next() ) {  
    cerveza = Menu.getString(1);  
    precio = Menu.getFloat(2);  
    /*algo con la cervez y el precio*/  
}
```



Módulos Persistentes

- **SQL/PSM:**
 - **Persistent Stored Modules**
 - Especifica la sintáxis y semántica del lenguaje para declarar y mantener **módulos persistentes** en un servidor SQL.
 - Extiende SQL haciéndolo un lenguaje **completo computacionalmente**.
 - Permite almacenar módulos (procedimientos y funciones) como elementos de un esquema de base de datos.



Módulos Persistentes

- **SQL/PSM** incluye:
 - Sentencias para dirigir el **flujo de control**.
 - **Asignación** del resultado de expresiones a **variables y parámetros**.
 - Especificación de **manejadores** de **condiciones** que permiten a las rutinas trabajar con diversas condiciones durante su ejecución.
 - Sentencias para condiciones de **señales**.
 - **Declaraciones** de **cursores y variables** locales.



Módulos Persistentes

- **Módulos** *[SQL-server module]*
 - Un módulo de servidor SQL es un objeto persistente definido en un esquema de base de datos.
 - Puede tener tablas temporales propias.
 - Contiene una o varias rutinas (procedimientos y funciones).

```
CREATE MODULE <nombre módulo>
  [ NAMES ARE <especificación conjunto caracteres> ]
  [ SCHEMA <nombre esquema> ] [ <ruta> ]
  [ <declaración de tabla temporal> ... ]
  [ <declaración de rutina> ... ]
END MODULE;

DROP MODULE <nombre módulo> ;
```



Módulos Persistentes - Rutinas

- **Rutinas** *[SQL-invoked routine]*
 - Pueden ser **procedimientos** o funciones.

```
CREATE PROCEDURE <nombre procedimiento>
  <lista parámetros> <características varias> <cuerpo>;

<lista parámetros> ::= [ <modo> ] [ <nombre> ] <tipo> [ RESULT ]
<modo> ::= { IN | OUT | INOUT }
<tipo> ::= <tipo de dato> [ AS LOCATOR ]
```



Módulos Persistentes - Rutinas

- **Rutinas** *[SQL-invoked routine]*

- Pueden ser procedimientos o **funciones**.
 - Los **métodos** son tipos especiales de funciones.

CREATE FUNCTION <nombre función>

<lista parámetros> <retorno> <características varias>

[<dispatch>] <designador método> <cuerpo>;

<retorno> ::= **RETURNS** { <tipo dato> | **TABLE** <lista columnas> }

<lista columnas> ::= (<nombre> <tipo dato> [, ...])



Módulos Persistentes - Rutinas

- **Ejemplo de Procedimiento**

- Dados dos argumentos, c y p, añade una fila a la tabla **Ventas(bar, cerveza, precio)** con bar='Gambrinues', cerveza=c y precio=p.

```
CREATE PROCEDURE Alta (IN c CHAR(20), IN p REAL)
```

```
INSERT INTO Ventas VALUES('Gambrinus', c, p);
```

- Uso del procedimiento:

```
CALL Alta ('Heineken', 2);
```



Módulos Persistentes - Rutinas

- El **cuerpo** de un **procedimiento o función** incluye una lista de **sentencias SQL ejecutables**:
 - Declaraciones de variables (DECLARE).
 - Definición y manipulación del esquema (CREATE, DROP).
 - Manipulación de datos (INSERT, UPDATE, DELETE, SELECT).
 - Control.
 - Llamadas y retornos (CALL, RETURN).
 - Agrupamiento (BEGIN .. END).
 - Asignación.
 - Bifurcación (CASE, IF, LEAVE).
 - Iteración (ITERATE, LOOP, WHILE, REPEAT, FOR).
 - Diagnóstico (SIGNAL, RESIGNAL).
 - Otras (transacciones, sesiones, conexiones, sql dinámico, ...).



Módulos Persistentes - Rutinas

- **Declaración de Variables.**

DECLARE <lista nombres variables> <tipo dato> [**DEFAULT**
<valor>]

- **Ejemplos**

```
DECLARE a,b INTEGER DEFAULT 0;
```

```
DECLARE cerveza CHAR(20);
```



Módulos Persistentes - Rutinas

- **Ejemplo de Función.** Dado el nombre de un cliente, devolver la cantidad de cuentas de las cuales es propietario.

```
CREATE FUNCTION nro_cuentas (nombre_cliente VARCHAR(20))
  RETURNS INTEGER
  BEGIN
    DECLARE a INTEGER;
    SELECT COUNT ( * ) INTO a FROM Depositos
    WHERE Depositos.nombre_cliente = nombre_cliente;
    RETURN a;
  END
```

- **Uso:** Nombre y dirección de los clientes con más de una cuenta.

```
SELECT nombre_cliente, direccion FROM Clientes
  WHERE nro_cuentas (nombre_cliente) > 1;
```



Módulos Persistentes - Rutinas

- **Ejemplo de Procedimiento.** Dado el nombre de un cliente, devolver la cantidad de cuentas de las cuales es propietario.

```
CREATE PROCEDURE nro_cuentasp (IN nombre_cliente VARCHAR(20),
  OUT a INTEGER)
  BEGIN
    SELECT COUNT(*) INTO a FROM Depositos
    WHERE Depositos.nombre_cliente = nro_cuentasp.nombre_cliente
  END
```

- **Uso:**

```
DECLARE ac integer;
CALL nro_cuentasp( 'Jorge Diaz', ac);
```



Módulos Persistentes - Rutinas

- Bloques

- Permiten agrupar sentencias como si fueran una.
- Las sentencias se separan entre sí con ";".

```
[ <etiqueta inicio> . ] BEGIN [ [ NOT ] ATOMIC ]  
  [ <lista declaraciones locales> ] [ <lista cursores locales> ]  
  [ <lista manejadores locales> ]  
  <sentencia SQL> [ ; ... ]  
  END [ <etiqueta fin> ]
```



Módulos Persistentes - Rutinas

- Sentencias

- RETURN <expresión>
 - Establece el valor de retorno de una función.
 - Al contrario que C, la ejecución de la función no concluye.
- DECLARE <nombre> <tipo>
 - Declarar variables locales.
- SET <variable> = <expresión>
 - Asignación
- <etiqueta> .
 - Identificar una sentencia por una etiqueta para luego poder referirla o saltar a ella (con LEAVE <etiqueta>).



Módulos Persistentes – Rutinas

- **SQL:2003** permite que el resultado retornado (return) por una función sea un tipo de dato o una tabla (**función de tabla**).

```
<retorno> ::= RETURNS { <tipo dato> | TABLE <lista columnas> }  
<lista columnas> ::= ( <nombre> <tipo dato> [, ... ] )
```

- **Ejemplo de función de tabla:** Cuentas de un cierto cliente.

- **Uso:**

```
SELECT * FROM cuentas_cli('Jose Diaz');
```



Módulos Persistentes – Rutinas

- **Ejemplo de función de tabla:** Cuentas de un cierto cliente.

```
CREATE FUNCTION cuentas_cli (nombre_cli CHAR(20)  
    RETURNS TABLE (nro_cuenta CHAR(10), sucursal  
    CHAR(15),  
                    saldo NUMERIC(12,2) )  
RETURN TABLE  
    (SELECT nro_cuenta, sucursal , saldo FROM cuentas c  
    WHERE EXISTS (  
        SELECT * FROM depositos d  
        WHERE d.nombre_cli = cuentas.nombre_cli  
        AND d.nro_cuenta = c.nro_cuenta ) );
```



Módulos Persistentes – Rutinas

- SQL:2003 permite usar **funciones y procedimientos externos**, escritos en lenguajes diferentes a SQL (C, C++, COBOL; ..).

```
CREATE PROCEDURE contar_cuentas_pr ( IN nombre_cli  
    VARHCAR(20), OUT cant INTEGER)  
LANGUAGE C  
EXTERNAL NAME ' /usr/avi/bin/contar_cuentas_pr';
```



Módulos Persistentes - Estructuras de Control

- Sentencias – Bifurcaciones

```
IF <condición> THEN <lista sentencias>  
    [ ELSEIF <condición> THEN <lista sentencias> ... ]  
    [ ELSE <lista sentencias> ]  
END IF
```

puede ser una
expresión condicional

```
CASE <expresión>  
    WHEN <lista operandos> THEN <lista sentencias>  
    [ ... ]  
    [ ELSE <lista sentencias> ]  
END CASE
```



Módulos Persistentes - Estructuras de Control

- Sentencias – Ejemplo de Bifurcaciones

- Crear una función que clasifica un bar en 'abandonado', 'medio' o 'popular' según sea frecuentado por menos de 100 clientes, entre 100 y 199 o más, respectivamente. Usar la tabla `Frecuentar`(`bebedor`, `bar`) para ello.

```
CREATE FUNCTION RatioBar (IN b CHAR(20) )
  RETURNS CHAR(15)
  DECLARE ncli INTEGER;
  BEGIN
    SET ncli = (SELECT COUNT(*) FROM Frecuentar
                WHERE bar = b);
    IF ncli < 100 THEN RETURN 'abandonado'
      ELSEIF cust < 200 THEN RETURN 'medio'
      ELSE RETURN 'popular'
    END IF;
  END;
```



Módulos Persistentes - Estructuras de Control

- Sentencias - Iteraciones

```
[ <etiqueta inicio> : ] LOOP
  <lista sentencias>
END LOOP [ <etiqueta fin> ]
```

Usar `LEAVE <etiqueta inicio>` para salir desde dentro de un bucle `LOOP`

```
[ <etiqueta inicio> : ] WHILE <condición> DO
  <lista sentencias>
END WHILE [ <etiqueta fin> ]
```

`ITERATE <etiqueta>`

- Termina una iteración.
- La etiqueta debe corresponder a la de inicio del bucle.



Módulos Persistentes - Estructuras de Control

- Sentencias - Iteraciones

```
[ <etiqueta inicio> : ] REPEAT
```

```
  <lista sentencias>
```

```
  UNTIL <condición>
```

```
END REPEAT [ <etiqueta fin> ]
```

```
[ <etiqueta inicio> : ] FOR [ <variable iterador> AS ]
```

```
  [ <nombre cursor> [ <sensitividad cursor> ] CURSOR FOR ]
```

```
  <especificación cursor>
```

```
  DO <lista sentencias>
```

```
END FOR [ <etiqueta fin> ]
```

Los cursores se presentan más tarde



Módulos Persistentes - Estructuras de Control

- Sentencias – Ejemplo de Iteraciones

```
DECLARE n INTEGER DEFAULT 0;
```

```
WHILE n < 10 DO
```

```
  SET n = n + 1
```

```
END WHILE;
```

```
REPEAT
```

```
  SET n = n - 1
```

```
UNTIL n = 0
```

```
END REPEAT;
```



Módulos Persistentes - Estructuras de Control

- Sentencias – Ejemplo de Iteraciones.
 - Calcular las ventas totales del departamento de 'Oriente'.

```
DECLARE n INTEGER DEFAULT 0;
FOR r AS
    SELECT importe FROM Ventas
    WHERE depart = 'Oriente'
DO
    SET n = n + r.importe
END FOR;
```



Módulos Persistentes – Excepciones

- En **SQL:2003** el control de excepciones se realiza mediante el **manejo de condiciones** declarando condiciones (*conditions*) y sus manejadores (*handlers*):

```
DECLARE <nombre condición> CONDITION [ FOR <valor sqlstate> ]
```

```
DECLARE <tipo manejador> HANDLER FOR <lista valores condición>
    <acción manejador>
```

<valor sqlstate> ::= SQLSTATE [VALUE] <string literal>

<tipo manejador> ::= { CONTINUE | EXIT | UNDO }

<lista valores condición> ::= <valor condición> [, ...]

<valor condición> ::= { <valor sqlstate> | <nombre condición> |
SQLEXCEPTION | SQLWARNING | NOT FOUND }



Módulos Persistentes – Excepciones

- En **SQL:2003** el control de excepciones se realiza ... y señalando las condiciones (*signal*):

```
SIGNAL {<nombre condición> | <valor sqlstate> }  
[ SET <item información de señal> [ , ... ] ]
```

```
<item información de señal> ::=  
    <nombre item condición> = <valor simple>
```

```
RESIGNAL [ {<nombre condición> | <valor sqlstate> } ]  
[ SET <item información de señal> [ , ... ] ]
```



Módulos Persistentes – Excepciones

- **Ejemplo** de manejo de excepciones.

```
DECLARE sin_stock CONDITION  
DECLARE EXIT HANDLER FOR sin_stock  
BEGIN  
...  
.. SIGNAL sin_stock  
END
```

El manejador es EXIT.

Cuando es señalada la condición sin_stock (la ejecución llega a dicha sentencia) se activo el manejador produciendo la salida (exit) del bloque de ejecución BEGIN-END.



Módulos Persistentes – Consultas

- En módulos y rutinas PSM no se pueden realizar consultas SELECT iguales que las utilizadas en SQL interactivo.
- En su lugar existen tres alternativas:
 - a. Consultas que devuelven un único valor situadas en el lado derecho de una **asignación**.
 - b. Una orden **SELECT .. INTO** (se exige devolver una única fila de resultados).
 - c. **Cursores**.



Módulos Persistentes – Consultas

- Ejemplo de asignación con SELECT monovalor.
 - Usando la variable local p y la tabla **Ventas(bar, cerveza, precio)**, guardar en p el precio al que el bar 'Gambrinus' tiene la cerveza 'Mahou':

```
SET p = (SELECT precio FROM Ventas
        WHERE bar = 'Gambrinus' AND
              cerveza = 'Mahou');
```



Módulos Persistentes – Consultas

- Ejemplo con `SELECT .. INTO <variable>`.
 - El mismo caso de antes:

```
SELECT precio INTO p FROM Ventas
      WHERE bar = 'Gambrinus' AND
      cerveza = 'Mahou';
```



Módulos Persistentes – Cursores

- Un **cursor** es, básicamente, una variable que contiene la lista de filas resultado de una consulta.
- Es el mecanismo utilizado en SQL para superar el **desajuste de impedancia** entre el manejo de datos **navegacional** (ir a registro n, saltar al siguiente, ...) y el manejo de datos **relacional** (operaciones sobre conjuntos de filas que dan nuevos conjuntos de filas).



Módulos Persistentes – Cursores

- Declaración de un Cursor.

```
DECLARE <nombre> [ <sensibilidad> ] [ <desplazabilidad> ]  
CURSOR [ <permanencia> ] [ <retornabilidad> ]  
FOR <consulta> [ ORDER BY <lista orden> ]  
FOR { READ ONLY | UPDATE [ OF <lista columnas> ] }
```

<sensibilidad> ::= { SENSITIVE | INSENSITIVE | ASENSITIVE }

<desplazabilidad> ::= { SCROLL | NO SCROLL }

<permanencia> ::= { WITH HOLD | WITHOUT HOLD }

<retornabilidad> ::= { WITH RETURN | WITHOUT RETURN }



Módulos Persistentes – Cursores

- Opciones al declarar cursores.
- Sensibilidad:
 - INSENSITIVE => El cursor usa una copia temporal de los datos. Cambios en los datos originales no son visibles (tipo snapshot).
 - SENSITIVE => Cambios son visibles.
 - ASENSITIVE => la visibilidad de los cambios depende de la implementación.
- Desplazabilidad:
 - SCROLL => todas las opciones para navegar (adelante, atrás, saltos) están disponibles.
 - NO SCROLL => solo se puede leer la siguiente fila.
- Permanencia:
 - WITH HOLD => Al salir de una transacción y comenzar la siguiente el cursor permanece abierto.
- Retornabilidad:
 - WITH RETURN => Si está declarado dentro de una rutina y no se cierra en ella, el cursor se devuelve como valor retornado por la rutina.



Módulos Persistentes – Cursores

- **Abrir un Cursor.**
 - La consulta definida en el cursor es evaluada y el cursor apunta a la primera fila de resultados.

OPEN <nombre cursor>

- **Cerrar un Cursor.**

CLOSE <nombre cursor>



Módulos Persistentes – Cursores

- **Navegación por un Cursor.**
 - Posiciona el cursor en una fila específica de resultados y recupera los valores de dicha fila.

FETCH [[<orientación>] **FROM**] <nombre cursor> **INTO**
<lista destinos>

<orientación> ::= { **NEXT** | **PRIOR** | **FIRST** | **LAST** |
{ **ABSOLUTE** | **RELATIVE** } <valor sencillo> }

- <lista destinos> representa una lista separada por comas de parámetros, columnas, arrays o variables embebidas.



Módulos Persistentes – Cursores

- Ejemplo.

```
CREATE PROCEDURE CervezasGambrinus()
  DECLARE c CHAR(20);
  DECLARE p INTEGER;
  DECLARE notFound CONDITION FOR SQLSTATE '02000';
  DECLARE listacervezas CURSOR FOR SELECT cerveza, precio FROM Ventas
    WHERE bar='Gambrinus' ORDER BY cerveza;

BEGIN
  OPEN listacervezas;
  recorrer: LOOP
    FETCH NEXT FROM listacervezas INTO c,p;
    IF notFound THEN LEAVE recorrer END IF;
    ... /* proceso con cada fila de resultados
  END LOOP;
  CLOSE listacervezas;
END
```



Módulos Persistentes – Cursores

- Posicionamiento para UPDATE/DELETE.

- Los cursores también se pueden emplear para localizar una fila en una tabla con vistas a su modificación (**UPDATE posicionado**) o eliminación (**DELETE posicionado**).

- Estas sentencias no usan la cláusula WHERE normal con predicado, sino un cursor.
- EL cursor debe estar abierto y posicionado en una fila usando previamente una sentencia FETCH.
- El cursor debe operar sobre un conjunto de resultados modificable y tener activada la opción FOR UPDATE.

- UPDATE <tabla> SET ... WHERE CURRENT OF <cursor>

- DELETE FROM <tabla> WHERE CURRENT OF <cursor>



Módulos Persistentes – Implementaciones

- Los fabricantes de SGBD relacionales han desarrollado lenguajes SQL programáticos , incluso antes de que existiera el estándar SQL/PSM.
 - Existen diferencias entre el estándar y las implementaciones de cada fabricante.
- Las principales implementaciones comerciales, de mayor a menor cumplimiento del estándar, son:
 - **SQL Procedural Language** (IBM DB2)
 - **PL/SQL** (ORACLE)
 - **Transact-SQL** (Microsoft SQL Server)



SQL Embebido

- La parte 2 (*Foundation*) del **SQL:2003** incluye el SQL Embebido para los siguientes lenguajes:
 - Ada
 - C
 - COBOL
 - FORTRAN
 - MUMPS
 - Pascal
 - PL/I
- La parte 10 (*SQL/OLB*) lo amplia para **Java**.



SQL Embebido

- Una **sentencia de SQL embebido** es una sentencia SQL escrita en un lenguaje anfitrión.
 - Se distingue porque está precedida de una etiqueta especial (**prefijo**).
 - Opcionalmente, también acaba con un **terminador** también especial.

<prefijo> <sentencia SQL> [<terminador>]

<prefijo> ::= { EXEC SQL | &SQL(}

<terminador> ::= { END-EXEC | ; |) }



SQL Embebido

- Un **precompilador** convierte las sentencias SQL en sus equivalentes llamadas a procedimientos en el formato del lenguaje anfitrión.
 - La identificación de los fragmentos de código SQL se hace en base a las etiquetas especiales (prefijo y terminador).
 - EXEC SQL <sentencia SQL> ;
 - EXEC SQL <sentencia SQL> END-EXEC
 - &SQL(<sentencia SQL>)



SQL Embebido

- Las mismas etiquetas también sirven para delimitar una **sección de declaraciones SQL-embebido**.

<prefijo> **BEGIN DECLARE SECTION** [<terminador>]

[**SQL NAMES ARE** <conjunto caracteres>]

[<definición variable> ...]

<prefijo> **END DECLARE SECTION** [<terminador>]

<definición variable> ::= { <definición variable lenguaje L>
| :<identificador lenguaje L> }

L = {Ada, C, COBOL, FORTRAN, MUMPS, Pascal, PL/I}



SQL Embebido

- **Variables Compartidas.**
 - Para conectar SQL y el programa en el lenguaje anfitrión ambos deben compartir algunas variables.
EXEC SQL BEGIN DECLARE SECTION;
 <declaraciones lenguaje anfitrión>
EXEC SQL END DECLARE SECTION;
 - Cuando se usan en SQL se preceden de ':'.
■ En el lenguaje anfitrión se usan de forma normal.



SQL Embebido

- Se pueden declarar **condiciones** para el **manejo de excepciones**.

WHENEVER <condición> <acción>

<condición> ::= { **SOLEXCEPTION** | **SQLWARNING** | **NOT FOUND** |
SQLSTATE (<XX> [, <XXX>]) | **CONSTRAINT** <nombre> }

<acción> ::= { **CONTINUE** | { **GOTO** | **GO TO** } <destino> }
<destino> ::= { <etiqueta> | <entero> }



SQL Embebido

- **Ejemplo de SQL embebido en C.**
 - Con la ya conocida tabla **Ventas(bar,cerveza,precio)**.

```
EXEC SQL BEGIN DECLARE SECTION;  
    char vbar[21], vcerveza[21];  
    float vprecio;
```

Nota: string de
21 para los 20
caracteres +
marca de fin

```
EXEC SQL END DECLARE SECTION;  
    /* obtener valores de bar y cerveza de alguna manera */  
EXEC SQL SELECT precio INTO :vprecio FROM Ventas  
    WHERE bar = :vbar AND cerveza = :vcerveza;  
    /* hacer algo con el precio */
```



SQL Embebido

- **Consultas.**
- SQL Embebido tiene las mismas limitaciones y opciones para consultas que las ya vistas para PSM:
 - Usar **SELECT-INTO** para una consulta que seguro devuelve una única fila de resultados.
 - Usar **Cursores**:
 - EXEC SQL DECLARE c CURSOR FOR <consulta>;
 - EXEC SQL OPEN CURSOR c;
 - EXEC SQL CLOSE CURSOR c;
 - EXEC SQL FETCH c INTO <variable(s)>;



SQL Embebido

- **Ejemplo de Consulta con Cursor.**

```
EXEC SQL BEGIN DECLARE SECTION;
    char vcerveza[21]; float vprecio;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c CURSOR FOR
    SELECT cerveza, precio FROM Ventas
    WHERE bar = 'Gambrinus';
EXEC SQL OPEN CURSOR c;
while(1) {
    EXEC SQL FETCH c INTO :vcerveza, :vprecio;
    if (NOT FOUND) break;
    /* hacer algo con cerveza y precio */
}
EXEC SQL CLOSE CURSOR c;
```

La declaración del cursor va fuera de la sección de declaraciones



SQL Embebido

- **Ejemplo de Consulta con Cursor.**
 - Encontrar nombres y ciudades de los clientes que tengan un saldo mayor de una cierta cantidad VM en alguna de sus cuentas. El valor de VM está en una variable compartida.

EXEC SQL

```
DECLARE c CURSOR FOR
  SELECT Depositos.nombre_cli, ciudad_cli
  FROM Depositos, Clientes, Cuentas
  WHERE Depositos.nombre_cli = Clientes.nombre_cli
  AND Depositos.nro_cta = Cuentas.nro_Cta
  AND Cuentas.saldo > :vm
```

END_EXEC



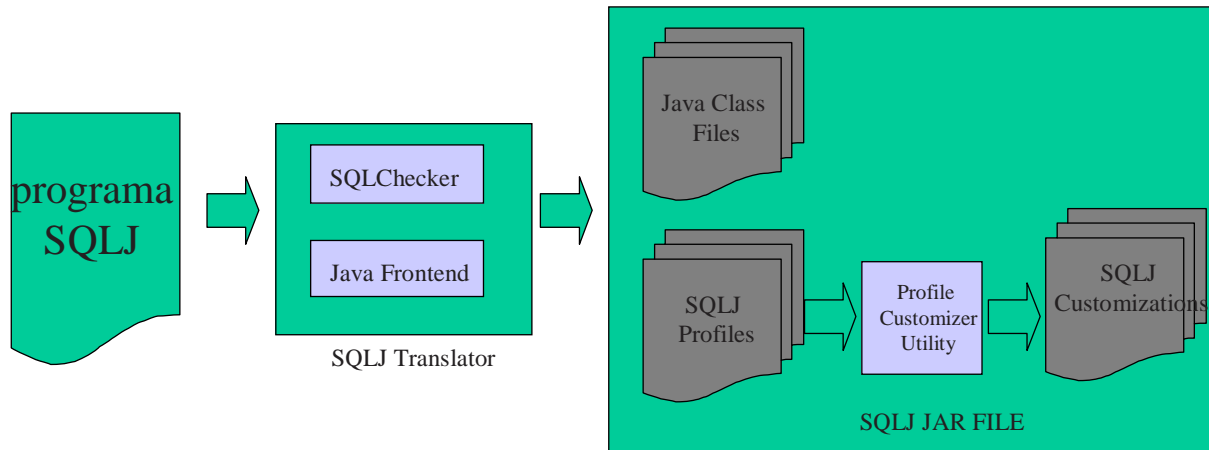
SQL Embebido - SQLJ

- **SQLJ:**
 - Permite sentencias **SQL embebidas** de forma **estática** en programas escritos en Java.
 - Su nombre oficial (parte 10 de SQL:2003) es **SQL/OLB** (*Object Language Bindings*).
 - Provee mecanismos para asegurar la portabilidad binaria de las aplicaciones SQLJ resultantes.
 - Especifica una lista de paquetes Java y sus clases y métodos.
 - Puede coexistir en una misma aplicación con JDBC (dinámico).



SQL Embebido - SQLJ

- **Entorno de Uso de SQLJ:**
 - Los vendedores de SGBD proveen versiones para instalar binarios (perfiles) SQLJ en las bases de datos.
 - Los binarios SQLJ funcionan con cualquier driver **JDBC**.



SQL Embebido - Dinámico

- El **SQL Dinámico** es una extensión del SQL Embebido para incorporar:
 - Disponer de un "descriptor area" para **comunicar** entre el anfitrión y el agente SQL.
 - **Ejecutar sentencias SQL**, incluida la preparación previa.
- ¿Por qué es necesario?
 - Porque es frecuente que las consultas no sean conocidas de forma exacta hasta el momento de su ejecución.
 - Utilizan **parámetros** (?) cuyos valores son provistos de forma interactiva por el usuario.



SQL Embebido - Dinámico

- La llamada **SQL Descriptor Area** (SQLDA, área de descriptores SQL) contiene la información (identificadores, tipos y códigos) para:
 - **Ejecución inmediata** de sentencias (preparar y ejecutar una sola vez).
 - **Asignar o cancelar espacio** para un descriptor de sentencia.
 - **Establecer descriptor** de sentencia y **recuperarlo**.
 - **Preparar** (optimizar) una sentencia para su ejecución.
 - **Cancelar** sentencias preparadas.
 - Obtener una **descripción** de los parámetros dinámicos de **entrada** para una sentencia preparada.
 - Obtener una **descripción** de las columnas de una sentencia select dinámica; o una descripción de los parámetros dinámicos de **salida** para el resto de sentencias.
 - **Ejecutar** una sentencia.



SQL Embebido - Dinámico

- **Ejecutar una sentencia.**
 - Previamente preparada:
 - **EXECUTE** <nombre sentencia> [<resultados salida>]
[<parámetros entrada>]
<resultados salida> ::= { **INTO** <lista de argumentos> |
INTO [**SQL**] <descriptor> }
 - Cláusula para **suministrar valores de salida**.
 - <parámetros entrada> ::= { **USING** <lista de argumentos> | <descriptor> }
 - Cláusula para **suministrar valores de entrada**.
 - Preparar y ejecutar una sola vez:
 - **EXECUTE IMMEDIATE** <variable con sentencia SQL>



SQL Embebido - Dinámico

- SQL Dinámico incorpora algunas otras sentencias nuevas o cambiadas frente al SQL Embebido estático.
 - **SET [SQL] DESCRIPTOR ...**
 - Poner información en un área de descriptores.
 - **PREPARE ... FROM**
 - Preparar una sentencia para su ejecución.
 - **OPEN <cursor> [<parámetros entrada>]**
 - Asociar parámetros dinámicos de entrada a una especificación de cursor y abrir el cursor.
 - **FETCH ... <cursor> [<resultados salida>]**
 - Localizar una fila en un cursor y guardar los resultados en la lista de resultados de salida.



SQL Embebido - Dinámico

- Ejemplo.

```
EXEC SQL BEGIN DECLARE SECTION;
    CHAR vconsulta[MAX_LENGTH];
EXEC SQL END DECLARE SECTION;
while(1) {
    /* preguntar al usuario la consulta a realizar y
    concatenarla */
    EXEC SQL PREPARE q FROM :vconsulta;
    EXEC SQL EXECUTE q;
}
```



SQL Embebido - Dinámico

- Ejemplo con un parámetro de entrada.

```
EXEC SQL BEGIN DECLARE SECTION;
  CHAR * vconsulta = "UPDATE cuentas
  SET saldo = saldo * 1.05 WHERE nro_cta = ?";
  CHAR vcuanta[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE p FROM :vconsulta;
vcuanta = "A-101";
EXEC SQL EXECUTE p USING :vcuanta;
```



SQL Embebido - Dinámico

- Ejemplo con varios parámetros de entrada.

```
EXEC SQL BEGIN DECLARE SECTION;
  CHAR * vconsulta = "UPDATE cuentas
  SET saldo = saldo * 1.05 WHERE nro_cta = :x AND
  provincia= :y";
  CHAR vcuanta[10];
  CHAR vpro[3];
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE p FROM :vconsulta;
vcuanta = "A-101";
vpro = "39"
EXEC SQL EXECUTE p USING :vcuanta, :vpro;
```



SQL Embebido – SQLJ vs JDBC

Consulta mono-fila	
SQLJ estándar estático	<pre>#sql [ctx] { SELECT MAX(SALARY), AVG(SALARY) INTO :maxSalary, :avgSalary FROM DSN8710.EMP };</pre>
JDBC no estándar dinámico	<pre>PreparedStatement stmt = conn.prepareStatement("SELECT MAX(SALARY), AVG(SALARY)" + " FROM DSN8710.EMP"); rs = stmt.executeQuery(); if (!rs.next()) { // Error -- no rows found } maxSalary = rs.getBigDecimal(1); avgSalary = rs.getBigDecimal(2); if (rs.next()) { // Error -- more than one row found } rs.close(); stmt.close();</pre>



SQL Embebido – SQLJ vs JDBC

Inserción	
SQLJ estándar estático	<pre>#sql [ctx] { INSERT INTO DSN8710.EMP (EMPNO, FIRSTNAME, MIDINIT, LASTNAME, HIREDATE, SALARY) VALUES (:empno, :firstname, :midinit, :lastname, CURRENT DATE, :salary) };</pre>
JDBC no estándar dinámico	<pre>stmt = conn.prepareStatement("INSERT INTO DSN8710.EMP " + "(EMPNO, FIRSTNAME, MIDINIT, LASTNAME, HIREDATE, SALARY) " + "VALUES (?, ?, ?, ?, CURRENT DATE, ?)"); stmt.setString(1, empno); stmt.setString(2, firstname); stmt.setString(3, midinit); stmt.setString(4, lastname); stmt.setBigDecimal(5, salary); stmt.executeUpdate(); stmt.close();</pre>



Otros Aspectos Avanzados

- **SQL:2003** tiene muchas **opciones avanzadas**.
 - Crear una tabla con el mismo esquema que otra existente:
`CREATE TABLE temp_cuentas LIKE cuentas;`
 - Subconsultas o modificaciones monovalor (1 columna, 1 fila) se pueden poner en cualquier sitio donde se requiere un único valor.
 - Subconsultas en la cláusula FROM para acceder a atributos de otras relaciones:
`SELECT C.nombre_cli, cant_ctas
FROM cliente C, LATERAL (
SELECT COUNT(*) FROM cuenta A WHERE
C.nombre_cli=C.nombre_cli);
AS este_cliente(cant_ctas);`



Otros Aspectos Avanzados

- **SQL:2003** tiene muchas **opciones avanzadas**.
 - El constructor MERGE permite procesamiento batch (por lotes) de modificaciones:
`MERGE INTO cuentas AS C
USING (SELECT * FROM fondos_recibidos AS F)
ON (C.nro_cta=F.nro_cta)
WHEN MATCHED THEN
UPDATE SET saldo=saldo+F.cantidad;`
 - siendo la tabla fondos_recibidos(nro_cta,cantidad) un registro de los depósitos que deben ser añadidos a las cuentas correspondientes de la tabla cuentas.



Otros Aspectos Avanzados – Recursividad

- SQL:2003 permite la definición de **vistas recursivas**.
 - Ejemplo: Encontrar todos los pares employee-manager, tal que el empleado reporta al manager de forma directa o indirecta (al manager del manager, al manager del manager del manager, etc.).

```
WITH RECURSIVE empl(employee_name, manager_name) AS (  
    SELECT employee_name, manager_name  
        FROM manager  
    UNION  
    SELECT manager.employee_name, empl.manager_name  
        FROM manager, empl  
        WHERE manager.manager_name = empl.employee_name)  
SELECT * FROM empl;
```



Otros Aspectos Avanzados – Recursividad

- Las vistas recursivas permiten escribir consultas que calculan **cierres transitivos**:
 - Si se incluye (a,b) y (b,c) => incluir (a,c)
 - En el ejemplo anterior, sin recursión no sería posible obtener los gestores a cualquier nivel, sino solo a un nivel concreto.
- El cierre transitivo se obtiene en sucesivos pasos (iteraciones recursivas).
 - En la consulta anterior, cada paso construye una versión extendido de la tabla empl a partir de su definición recursiva hasta obtener la versión final (fixed point).



Otros Aspectos Avanzados – Recursividad

Tabla manager

<i>employee_name</i>	<i>manager_name</i>
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Tabla empl

<i>Iteration number</i>	<i>Tuples in empl</i>
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)