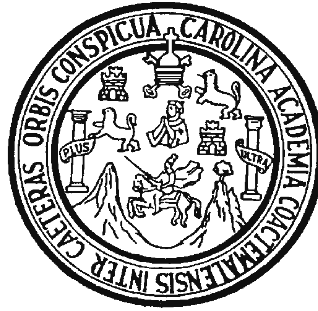
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA
ESCUELA DE INGENIERÍA EN CIENCIAS Y SISTEMAS

SISTEMAS DE BASES DE DATOS ACTIVAS: DISPARADORES Y REGLAS

Xiomara Carolina Vivar López
Asesorada por: Inga. Floriza Avila Pesquera

GUATEMALA, SEPTIEMBRE DE 2003

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA



FACULTAD DE INGENIERÍA

SISTEMAS DE BASES DE DATOS ACTIVAS: DISPARADORES Y REGLAS

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA

FACULTAD DE INGENIERÍA

POR

XIOMARA CAROLINA VIVAR LOPEZ
ASESORADA POR INGA. FLORIZA AVILA PESQUERA

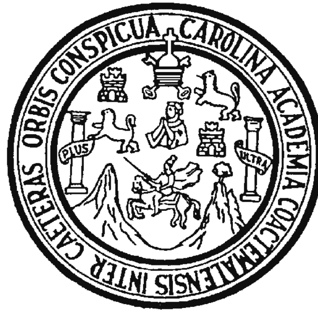
AL CONFERÍRSELE EL TÍTULO DE

INGENIERA EN CIENCIAS Y SISTEMAS

GUATEMALA, SEPTIEMBRE DE 2003

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA



NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Sydney Alexander Samuels Milson
VOCAL I	Ing. Murphy Olympo Paiz Recinos
VOCAL II	Lic. Amahán Sánchez Álvarez
VOCAL III	Ing. Julio David Galicia Celada
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazmina Vides Leiva
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Sydney Alexander Samuels Milson
EXAMINADOR	Ing. Jorge Luis Alvarez Mejía
EXAMINADOR	Ing. Edgar René Ornelis Hoil
EXAMINADOR	Ing. Claudia Liceth Rojas Morales
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

SISTEMAS DE BASES DE DATOS ACTIVAS: DISPARADORES Y REGLAS

Tema que me fuera asignado por la Coordinación de la Carrera de Ingeniería en Ciencias y Sistemas con fecha 26 de Febrero de 2002

Xiomara Carolina Vivar López

Universidad de San Carlos de Guatemala
Facultad de Ingeniería

Guatemala, Julio 28 de 2,003

Ingeniero
Carlos Azurdia
Coordinador de Privados y Revisión de Tesis
Escuela de Ciencias y Sistemas

Estimado Ingeniero:

Por medio de la presente, me permito informarle que he asesorado el trabajo de graduación titulado: **SISTEMAS DE BASES DE DATOS ACTIVAS: DISPARADORES Y REGLAS**, elaborado por el estudiante **Xiomara Carolina Vivar López**, y a mi juicio el mismo cumple con los objetivos propuestos para su desarrollo.

Agradeciéndole de antemano la atención que le preste a la presente, me suscribo de usted.

Atentamente,

Inga. Floriza Ávila Pesquera
Asesor de tesis

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas

Guatemala, 04 de Agosto de 2003

Ingeniero
Luis Alberto Vettorazzi España
Directo de la Carrera de Ingenieria
En Ciencias y Sistemas

Respetable Ingeniero Vettorazzi:

Por este medio hago de su conocimiento que he revisado el trabajo de graduación de la estudiante **XIOMARA CAROLINA VIVAR LÓPEZ**, titulado: **"SISTEMAS DE BASES DE DATOS ACTIVAS: DISPARADORES Y REGLAS"**, y que a mi criterio el mismo cumple con los objetivos propuestos para su desarrollo, según el protocolo.

Al agradecer su atención a la presente, aprovecho la oportunidad para suscribirme.

Atentamente,

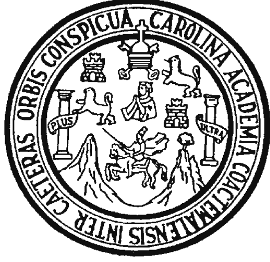
Ing. Carlos Alfredo Azurdia
Coordinador de Privados
y Revisión de Trabajos de Graduación

Universidad de San Carlos de Guatemala
Facultad de Ingeniería

El Coordinador de la Carrera de Ingeniería en Ciencias y Sistemas de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor con el visto bueno del revisor de tesis y del licenciado en Letras, al trabajo de graduación titulado**: SISTEMAS DE BASES DE DATOS ACTIVAS: DISPARADORES Y REGLAS** presentado por el estudiante universitario **Xiomara Carolina Vivar López,** aprueba el presente trabajo y solicita la autorización del mismo.

____
Ing. Luis Alberto Vettorazzi España
Coordinador
Ingeniería en Ciencias y Sistemas

Guatemala, Julio 30 de 2003



Universidad de San Carlos de Guatemala
Facultad de Ingeniería

El Decano de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Coordinador de la Escuela de Ingeniería en Ciencias y Sistemas, al trabajo de tesis titulado: **SISTEMAS DE BASES DE DATOS ACTIVAS: DISPARADORES Y REGLAS**, presentado por el estudiante universitario **Xiomara Carolina Vivar López**, procede a la autorización para la impresión de la misma.

IMPRÍMASE:

Ing. Sydney Alexander Samuels Milson
DECANO

Guatemala, Julio 30 de 2003

DEDICATORIA:

A DIOS
Gracias, Padre bueno, por mi familia.
Gracias por mis mayores que tantas cosas buenas me dejaron.
Gracias por mostrarme el camino hacia mi meta y permitirme alcanzarla.

A MIS PADRES
Clara Rosa López de Vivar
René Vivar Monroy
Infinitas gracias por todo su amor, gracias por brindarme su apoyo incondicional en los momentos difíciles.

A MI HERMANO
Juan René
Por apoyarme siempre incondicionalmente, e impulsarme a terminar mi carrera.

A MIS HERMANAS
Silma Eunise
Carol Jacqueline
Gracias porque puedo contar siempre con ustedes.

A MIS SOBRINOS Y PRIMOS
Espero ser un ejemplo para ellos.

A MIS ABUELOS

Lucrecia de Jesús Monroy Navas (QEPD)
Cristina Pérez Castro (QEPD)
Refugio Lopez (QEPD)
Flores sobre sus tumbas
Paulino Vivar Toledo

A MIS TÍAS Y TÍOS
Con cariño y respeto.

A MIS AMIGOS
A todos ellos, en especial a Ingrid Ovalle, Carina Orellana, Sergio Jiménez, gracias por ser mis amigos.

A MIS COMPAÑEROS
Blanca Castillo, Gladys Salazar, Betty Orozco, Mónica Dávila, Drixdel Ramos.

AGRADECIMIENTOS:

Agradecimiento especial a inga. Floriza Avila, por su decidida y desinteresada colaboración.

Gracias, a todas aquellas personas que me motivaron para terminar mi carrera.

Tuti, gracias por enseñarme que es mejor el triunfo y más grande la dicha, cuando se ha puesto todo el corazón para alcanzarlo.

ÍNDICE GENERAL

ÍNDICE DE ILUSTACIONES………………………………………………………. VI
LISTA DE SÍMBOLOS……………………………………………………………… VII
GLOSARIO…………………………………………………………………………… IX
RESUMEN……………………………………………………………………………. XIII
OBJETIVOS…………………………………………………………………………. XV
INTRODUCCIÓN…………………………………………………………………… XVI

1. INTRODUCCIÓN A LOS SISTEMAS DE BASES DE DATOS ACTIVAS
1.1 Historia de los sistemas de bases de datos activas……………………………. 1
1.2 ¿Qué son bases de datos activas?.. 1
1.3 Arquitectura básica de sistemas de bases de datos activas…………………… 2
1.3.1 Eventos………………………………………………………………….. 4
1.3.2 Condiciones…………………………………………………………… 6
1.3.3 Acciones………………………………………………………………. 6
1.3.4 Reglas ECA...………..……………….………………….……………… 7
1.3.5 Base de datos..……….……….……….…………….……….…….….…… 7
1.3.6 Sistema de administración de base de datos activa…..……....………….. 7
1.4 Reglas de las bases de datos activas…..…….……..…...….……….…..……….. 8
1.4.1 Lenguaje de la regla activa.…..……………...…...…..…….……...……….… 10
1.4.2 Sintaxis y semántica del lenguaje de la regla activa…….………..…..…. 10

1.4.3 Aspectos teóricos de las reglas activas………….……..…...……...…… 10
1.4.3.1 Detección de eventos….…….……..…..…….…….……...……. …. 11
1.4.3.2 Ejecución de la regla...…….……...…...…….…….……...…………. 11
1.4.3.3 Modelo de ejecución de reglas………...…...…..……...…………. 11
1.4.3.4 Modo de acoplamiento….…….……...…...…….…….……...………. 12
1.4.3.5 Resolución de conflicto….…….……...…...…….…….……...………. 13
1.5 Bases de datos activas vrs. bases de datos pasivas…….……...……….….. 13

2. SINTAXIS DEL LENGUAJE DE CDOL BASADO EN REGLAS

2.1 ¿qué es *Cdo*l? ……….……...…...…….……...…...…….……...…. ….. 15
2.2 Símbolos básicos..…….……...…...…….……...…...…….…….…… 15
2.3 Tipos……...…….…....…….……...…...…….……...……...……….….. 17
2.4 Construcciones básicas del modelo…….……...…...…….……...………... 18
2.5 Expresiones………….……...…...…….…...…...…….…….….……….…. 19
2.6 Lenguaje de consulta de la regla………...…...…….……...….……….…. 21
2.7 Sublenguaje de restricción….…….……...…...……..….……...….................. 22
2.8 Sublenguaje de actualización…...…...……….……...….…….…….……. 23
2.9 Lenguaje de la regla activa……....…...…….……...……...……….……. 24
2.10 idioma de definición de datos….…….……...…...…….……...…….…….. 25

3. SISTEMAS DE ADMINISTRACIÓN DE BASE DE DATOS ACTIVAS

3.1 Introducción..…….……...…...…….……...…...…….……....…………. 27
3.2 Sistemas de base de datos activas relacionales…...…..……….…….……… 27
3.2.1 *Órale*….…….……...…...…….……...…...…….……...…….……. 27
3.2.1.1 *Roke*……….……...…...…….……...…...…….……...…….… 28
3.2.1.2 Arquitectura *Roke*….…….……...…...…….…...…….…......… 28
3.2.1.3 *Arena*….…….……...…...…….……...…...…….……...…...… 31
3.2.1.4 Coordinación de procesos de negocio…….…….…...…...…...…... 38
3.2.1.5 Un caso práctico: proceso de atención de reclamos………....….… 42

3.2.2 *Db2*………………………………………………………………… 44

3.2.2.1 Sintaxis de *db2 create trigger*……………………………………… 44

3.2.2.2 Semántica de disparadores………………………………………… 45

3.2.2.3 Usando disparadores para implementar integridad referencial………… 46

3.2.2.4 Ejemplos de ejecución de disparadores…………………………… 47

3.2.3 Sql3………………………………………………………………… 47

3.2.3.1 Características activas en *sql3*……………………………………… 48

3.2.3.1.1 Disparadores en *sql3*…………………………………… …… 49

3.2.4 *Postgres*…………………………………………………………… 50

3.2.4.1 ¿Qué es *Postgres*? ………………………………………… …… 50

3.2.4.2 El sistema de reglas de *Postgres*…………………………… …… 51

3.2.4.2.1 ¿Qué es un árbol de consulta? ……………………………… 51

3.2.4.2.2 Las vistas y el sistema de reglas………………………… …… 53

3.2.4.2.2.1 Como trabajan las reglas de *select*……………… …… 54

3.2.4.2.2.2 El poder de las vistas en *Postgres*………………………… 57

3.2.4.2.3 Reglas y permisos………………………………………… …… 57

3.2.4.2.4 Reglas contra disparadores……………………………………… 59

3.2.4.3 Sintaxis de *create rule*……………………………………………… 60

3.3 Sistemas de base de datos activas orientadas a objetos

3.3.1 *Chimera*………………………………………………………………… 60

3.3.1.1 El modelo de *Chimera*…………………………………………… 61

3.3.1.1.1 Clases……………………………………………………… 61

3.3.1.1.2 Herencia…………………………………………………… 61

3.3.1.1.3 Vistas……………………………………………………… 62

3.3.1.1.4 Restricciones…………………………………………………… 62

3.3.1.2 El lenguaje de *Chimera*……………………………………… …… 63

3.3.1.3 Sintaxis y semántica de *Chimera define trigger*…………………… 63

3.3.1.4 Transacciones en *Chimera*……………………………………………… 64

3.3.2 *Hipac*……………………………………………………… 64
3.3.3 *Ode*……………………………………………………… 65
3.3.4 *Polyhedra*……………………………………………………… 66
3.3.4.1 Evitar la necesidad de mecanismos de concurrencia en base de datos… 69
3.3.4.1.1 Introducción……………………………………………… 69
3.3.4.1.2 Bases de datos activas…………………………………… 73
3.3.4.2 Desnormalización………………………………………………… 75
3.3.4.3 Ejemplo……………………………………………………… 76
3.3.4.3.1 La base de datos marriage………………………………… 76
3.3.4.3.2 El esquema………………………………………………… 76
3.3.4.3.3 El código activo…………………………………………… 77
3.3.4.3.4 Ejemplo de *Sql*…………………………………………… 79

4. METODOLOGÍA DE IDEA

4.1 Proyecto Idea……………………………………………………… 83
4.2 Estructura del proceso Idea……………………………………… 85
4.3 Diseño de la regla activa………………………………………… 87
4.4 Prototipo de regla activa………………………………………… 89
4.4.1 Propiedades de la ejecución de la regla activa……………… 89
4.4.2 Técnicas de modularización para el diseño activo de la regla…… 92
4.5 Implementación de reglas activas………………………………… 95

5. VENTAJAS Y DESVENTAJAS DE LAS BASES DE DATOS ACTIVAS

5.1 Ventajas……………………………………………………… 97
5.2 Desventajas…………………………………………………… 98

6. DISPARADORES

6.1 Historia de los disparadores en las bases de datos……..……...…...….………. 99
6.2 Facilidad en la especificación de eventos……………………………………....... 100
6.3 Ejemplos……..……...…...….……….……..……..……...…...….……….……...….... 101

CONCLUSIONES………………………………………………………………. 104
RECOMENDACIONES……………………………………………………………... 107
BIBLIOGRAFÍA………………………………………………………………… 108

ÍNDICE DE ILUSTRACIONES

FIGURAS

1	Arquitectura básica de una base de datos activa	3
2	DBMS activos vrs. DBMS pasivo	13
3	Arquitectura *Roke*	29
4	Detección de eventos	37
5	Arquitectura *Ruleflow*	0
6	Proceso de atención de reclamos	43
7	Estructura del proceso IDEA	86
8	Fin de la ejecución de la regla	89
9	Confluencia de la ejecución de la regla	90
10	iagrama de flujo de un disparador	100

TABLAS

I	Precedencia de operadores en los eventos	5
II	Símbolos de la gramática de *Cdol*	16
III	Reglas de integridad referencial	46

LISTA DE SÍMBOLO

ADBMS Sistema de administración de base de datos activa (*database management system active*)

ANSI Instituto Nacional Americano de Normas *(American National Standards Institute*)

API Interface de aplicación de programa (*Application Program Interface*).

ARENA Lenguaje de regla activa (Active *rule languaje*).

CDOL Lenguaje de objeto comprensivo declarativo (*Comprehensive, Declarative Object Language*)

DBA Administrador de base de datos (*Database administrator*)

DBMS Sistema de administración de base de datos (*database management system*)

DOOD Sistemas deductivos orientados a objetos (*Deductive Object Oriented systems*)

ECA Evento-Condición-Acción (*Event-Condition-Action*)

IR Integridad referencial

ISO Organización internacional para la estandarización (*International Organization for Standardization*)

LDD Lenguaje de definición de datos

RDMBS Sistema de administración de base de datos relacional (*Relational Database Management System*)

SQL Lenguaje de Consulta Estructurado (*Structured Query Language*).

GLOSARIO

Base de datos deductivas Una base de datos deductiva es en la que se puede derivar información a partir de la que se encuentra almacenada explícitamente. Como elementos constitutivos de una base de datos deductiva se encuentran los hechos, reglas de inferencia y las restricciones de integridad.

Catálogo Un componente de un diccionario de datos que contiene un directorio de los objetos de la DBMS así como los atributos de cada objeto.

Commit Los DBMS ofrecen sentencias especializadas para la gestión de transacciones, la nomenclatura habitual en bases de datos relacionales es: *COMMIT WORK*, finalizar una transacción*; ROLLBACK*, deshacerla.

Diccionario de datos Reúne la información sobre los datos almacenados en la base de datos.

Esquema La definición lógica y física de los elementos de los datos, características físicas e interrelaciones.

Interbloqueos — Los interbloqueos se producen cuando dos transacciones que acceden a una base de datos se bloquean mutuamente al intentar realizar un bloqueo exclusivo sobre los mismos recursos.

Integridad — Mantener la integridad de una base de datos es asegurarse de que los datos que contiene son correctos, evitando datos inconsistentes o erróneos de cualquier otro tipo

LDD — Lenguaje de definición de datos, se utiliza para crear y mantener la base de datos y los elementos que contiene a nivel externo, lógico e interno.

Modelo de datos — Es un conjunto de conceptos, reglas y convenciones que permiten describir y manipular los datos.

Modelo relacional — Propuesto por *Codd* a finales de los sesenta introduce la teoría de las relaciones en el campo de las bases de datos. En este modelo los datos se estructuran en tablas manteniendo la independencia de esta estructura lógica respecto al modo de almacenamiento u otras características físicas. Las tablas se manejan mediante operaciones de la teoría de conjuntos y el álgebra relacional.

Normalización Según el modelo relacional, las tablas deben definirse siguiendo una serie de reglas precisas para asegurarse de que no se producirán anomalías en la actualización de la base de datos. Para ello, es habitual que se necesite descomponer las tablas iniciales en otras más simplificadas que no presenten dichos problemas. Este proceso es lo que se conoce como normalización y es un método formalizado con diferentes niveles, a cada uno de los cuales se le llama forma normal.

Protocolo Una serie de convenciones que gobiernan las comunicaciones entre procesos.

Redundancia Se llama redundancia al hecho de que los mismos datos estén almacenados más de una vez en la base de datos. Las redundancias además de suponer un consumo de recursos de almacenamiento pueden llevar a situaciones en las que un dato se actualice en una de sus ubicaciones y en otra no y se pierda la integridad de la BD, por tanto deben evitarse.

Sistemas de *workflow* *Software* que permite la automatización de los procesos y de los flujos de trabajo donde documentos, información o tareas pasan de un trabajador a otro en orden a unas reglas y procedimientos.

SQL — *El SQL* es un lenguaje de alto nivel, no procedural, normalizado que permite la consulta y actualización de los datos de base de datos relacionales. Se ha convertido en el estándar de acceso a base de datos relacionales.

Transacción — Conjunto de modificaciones sobre una BD que son una unidad inseparable. Es decir, si se realiza alguna de las modificaciones deben realizarse todas, en caso contrario no debe realizarse ninguna.

RESUMEN

Un sistema de base de datos activa es simplemente una extensión de los sistemas pasivos más comunes de las bases de datos que están actualmente disponibles, por ejemplo las viejas versiones de *Orácle*. El foco actual dentro de los sistemas de la base de datos ha sido aumentar la tecnología de las bases de datos con las características nuevas para asistir al DBA, así como otros usuarios del sistema. En el capítulo "Introducción a los sistemas de bases de datos activos", se hace una descripción de la historia de los sistemas de administración de bases de datos activas, se define que son las bases de datos activas, se da la definición de regla activa, la sintaxis y semántica del lenguaje de la regla activa, se hace una comparación entre Sistemas de Bases de Datos Activas y sistemas de bases de datos pasivas.

CDOL es un lenguaje basado en las reglas de bases de datos activas, conocido como lenguaje de objeto comprensivo, declarativo que es una integración de la tecnología deductiva, orientada a objetos, y de Bases de Datos Activas. *CDOL* proporciona un sublenguaje para la expresión de datos derivados, restricciones, actualizaciones y reglas activas. En la parte titulada "Sintaxis del lenguaje de *CDOL* basado en reglas", se describe lo que respecta al lenguaje de *CDOL*.

Se ha definido que una Base de Datos Activa almacena la semántica de los datos además, de los propios datos. La siguiente parte "Sistemas de administración de base de datos activas", se describen como los diferentes sistemas de administración de base de datos activas, tanto relacionales, como orientados a objetos y relacionales orientados a objetos, manejan esta semántica, con el uso de disparadores y reglas activas. Estos DBMS son *Orácle, Db2, Postgres, Chimera, Hipac, Ode, Polyhedra*.

“Metodología IDEA”, un proyecto financiado por la Comunidad Europea, inició en junio de 1992 y terminó en abril de 1997. El objetivo fue desarrollar un prototipo de sistemas activos deductivos orientados a objetos y proporcionar la metodología y las herramientas para extender el uso de la tecnología activa de DOOD y de integrar de forma coherente los conceptos, metodología y herramientas necesarios para diseñar y desarrollar aplicaciones de bases de datos.

Los DBMS activos ofrecen ventajas sobre los DBMS tradicionales, en el penúltimo capítulo, se abarcan las ventajas y desventajas de las Bases de Datos Activas, y finalmente el capítulo 6 se describirán aspectos importantes de los disparadores.

OBJETIVOS

- **GENERAL**

En este trabajo, se quiere presentar las bases teóricas de los sistemas administradores de bases de datos activas, para establecer un antecedente de esta nueva tecnología de los sistemas de bases de datos.

- **ESPECÍFICO**

1. Introducir el concepto de bases de datos activas, conocer para qué tipo de aplicaciones se puedan usar.

2. Presentar la arquitectura básica de los sistemas de bases de datos activas.

3. Realizar una investigación de los Sistemas de Bases de Datos Activas relacionales y para los orientados a objetos.

4. Realizar el estudio de las reglas activas y los disparadores en los sistemas de bases de datos.

INTRODUCCIÓN

Los DBMS constituyen el núcleo fundamental del soporte lógico a los sistemas de información. Desde la aparición de los primeros DBMS comerciales en la década de los 60 hasta la actualidad, se han sucedido tres generaciones distintas de DBMS, DBMS relacionales, DBMS orientados a objetos, DBMS activos.

El avance tecnológico y la evolución en los Sistemas de Base de Datos han desarrollado una nueva tecnología denominada Sistema de administración de Base de datos Activas. Es necesario introducir los conceptos teóricos de éstos, ya que, actualmente en Guatemala es difícil encontrar información sobre los DBMS activos, aunque esta tecnología comenzó a conocerse a finales de los años 70 y se desarrolló el primer DBMS activo en 1987.

Los Sistemas de Administración de Bases de Datos Activas jugarán un papel muy importante en el campo de los futuros sistemas de información. Esta nueva tecnología ha tomado en los últimos años, gran interés por parte de los grupos de investigación alrededor del mundo. Se estima que las bondades de las bases de datos activas permitirán implementar novedosas aplicaciones.

Es importante hacer notar que estos sistemas de administración de base de datos, son la novedad en DBMS y ofrecen ventajas sobre los DBMS relacionales y por lo tanto no tardará mucho tiempo para que cobren importancia en los sistemas de información.

Los Sistemas de Administración de Bases de Datos Activas se utilizan en aplicaciones de sistemas expertos y bases de conocimientos que son área de inteligencia artificial. Así como en el comercio electrónico, y facilitan tareas como la integridad de la base de datos, y la seguridad.

El concepto sobre el cual están fundamentadas las Bases de Datos Activas es el de reglas activas, sentencias que determinan la manera como debe reaccionar la base de datos frente a situaciones expresadas en términos de eventos y/o condiciones. Las reglas en Sistemas de Bases de Datos Activas se pueden implementar por medio de reglas y disparadores.

Cuando se consulte este trabajo se podrá encontrar la información teórica de las bases de datos activas, y se abarcara información de Sistemas de Bases de Datos Activos con respecto a las reglas y disparadores que utilizan, así como un estudio de sus ventajas y desventajas

1. INTRODUCCIÓN A LOS SISTEMAS DE BASES DE DATOS ACTIVAS

1.1 Historia de los Sistemas de Bases de Datos Activas

Un Sistema de Base de Datos Activa es simplemente una extensión de los sistemas pasivos más comunes de las bases de datos que están actualmente disponibles, por ejemplo las viejas versiones de *Orácle*. El foco actual dentro de los sistemas de la base de datos ha sido aumentar la tecnología de las bases de datos con las características nuevas para asistir al DBA, así como otros usuarios del sistema.

La necesidad de crear un sistema más dinámico a hecho que surja las Bases de Datos Activas. La capacidad de construir una Base de Datos Activa ha estado disponible desde finales de los años 70 y a principio de los años 80. La tecnología no fue aceptada hasta mediados de los años 80, con el sistema *Hipac*.

Las Bases de Datos Activas son formas inteligentes de base de datos, ellas eliminan la necesidad de muchos de los subsistemas de aplicación que se requieren actualmente para traer bases de datos pasivas hasta estándares razonables de operación. Una Base de Datos Activa confía en un conjunto de reglas predefinidas, que pueden incorporar disparadores, condiciones y acciones. De esta manera es posible definir una Base de Datos Activa con las reglas que permiten que asuma el control de funciones tales como restricciones de integridad, vistas, seguridad.

Comúnmente en bases de datos pasivas la facilitación de sub-tareas es ocupada por la construcción de una capa de aplicación que interactúa recíprocamente con la base de datos. La capa de aplicación en bases de datos pasivas se puede implementar con herramientas de programación que el usuario utiliza para crear las aplicaciones de los clientes.

1.2 ¿Qué son Bases de Datos Activas?

Un Sistema de Base de Datos Activa puede reaccionar automáticamente a los acontecimientos. Esto está en contraste a los sistemas tradicionales de la base de datos que son pasivos en su comportamiento. Una Base de Datos Activa almacena la semántica de los datos, además de los propios. Una Base de Datos Activa utiliza reglas de evento-condición-acción (ECA, *Event-Condition-Action*). La ocurrencia de varios tipos de acontecimientos (por ejemplo, transiciones de la base de datos, acontecimientos del tiempo, y señales externas) acciona la evaluación de una condición y si la condición resulta verdadera, la acción se realiza. Las acciones se pueden implementar por medio de disparadores y reglas.

1.3 Arquitectura básica de Sistemas de Bases de Datos Activas

Básicamente un Sistema de Base de Datos Activa se encuentra compuesto por lo siguiente:

A) EVENTOS

- Eventos primitivos
  - -Eventos temporales
  - -Eventos definidos por los usuarios
  - -Eventos de la base de datos
- Eventos compuestos

B) CONDICIONES

C) ACCIONES

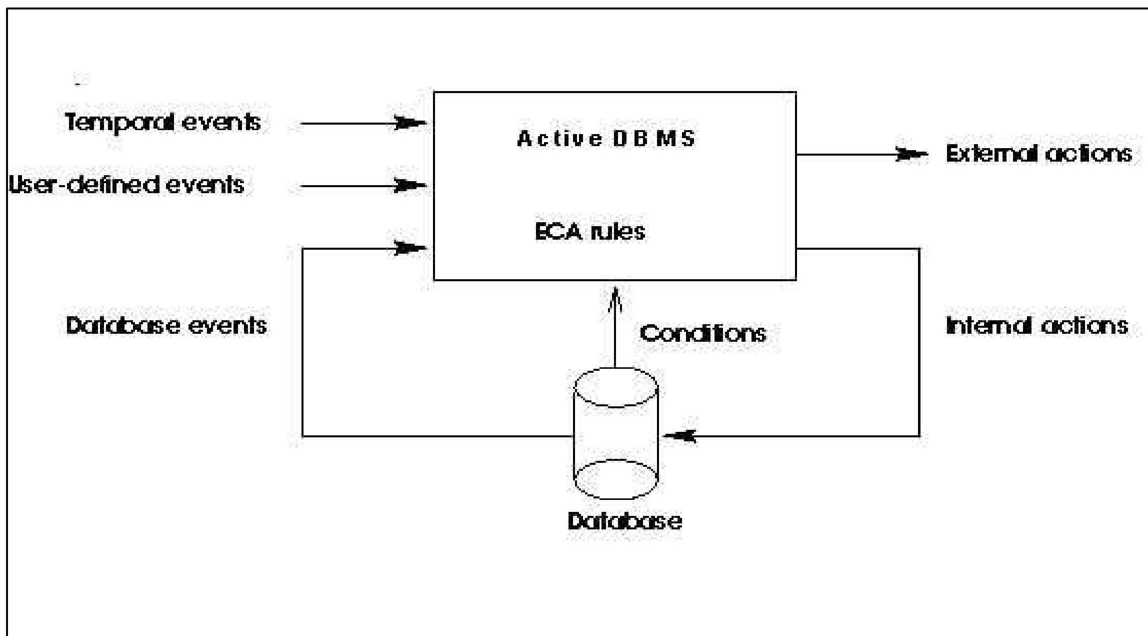
- Externas
- Internas

D) REGLAS ECA

E) BASE DE DATOS

La arquitectura básica de la una Base de Datos Activa se muestra en la figura 1.

Figura 1. Arquitectura básica de una Base de Datos Activa



1.3.1 Eventos

Un evento se concibe como una pareja {<Tipo de evento>, <Toc>}, donde el tipo de evento es la descripción o especificación del evento a detectar y toc (tiempo de ocurrencia) corresponde al punto en el tiempo cuando ocurre dicho tipo de evento.

Tipos de eventos pueden ser situaciones dentro de la base de datos o sucesos en el ambiente. En el caso de bases de datos relacionales, las operaciones *select, insert, delete* o *update* pueden ser consideradas como tipos de eventos, mientras que en bases de datos orientadas a objeto, éstos corresponden a la creación o borrado de objetos así como el llamado a métodos. El tiempo también puede ser considerado un tipo de evento interesante tal como un punto absoluto en el tiempo (“10 de julio de 1998") o puntos relativos o periódicos en el tiempo (“cada martes a las 10 a.m.").

Generalmente, un evento posee parámetros adicionales a su tiempo de ocurrencia tales como el identificador de la transacción que lo ocasionó, el identificador del usuario que inició la transacción o por ejemplo el nombre del método, identificador del objeto y argumentos en el llamado a un método en bases de datos orientadas a objetos. Estos parámetros pueden ser referenciados tanto en la condición como en la acción de la regla. Se denomina historia de eventos al conjunto de todos los eventos sucedidos en el tiempo. La historia inicia desde el momento en que se define el primer tipo de evento en la base de datos. Los eventos se clasifican en:

Eventos primitivos: Corresponden a ocurrencias de tipos de eventos que pueden ser detectados directamente por la base de datos.

Eventos compuestos: se definen como expresiones de eventos utilizando un conjunto de operadores de eventos tales como: disyunción “|”, conjunción “&” el operador secuencia “;”.

Suponiendo dos eventos primitivos A y B, A&B ocurre cuando A y B ocurren, A|B ocurre cuando A o B ocurren y A; B ocurre cuando ocurre A y luego ocurre B. Se han propuesto más operadores de eventos (y su precedencia) con el ánimo de aumentar la riqueza de eventos compuestos.

El tiempo de ocurrencia de un evento compuesto es igual al del evento primitivo que ocasionó la detección del evento compuesto. La precedencia de los operadores determina el orden en que se evalúa la semántica de un evento compuesto. En la tabla I, se muestra el orden de la precedencia de operadores que se evalúan en los eventos.

Tabla I. Precedencia de operadores en los eventos

ORDEN	OPERADOR
1	NOT
2	AND
3	SEQ
4	OR

Los eventos primitivos se clasifican en eventos de la base de datos, temporales y explícitos.

- Los eventos de la base de datos corresponden a las operaciones de la base de datos, por ejemplo *insert, update, delete, select* (en el modelo relacional) y los métodos (en el modelo orientado a objetos).

- Los eventos temporales se clasifican en absoluto y relativo. Temporal Absoluto*:* se definen como eventos de tiempo que estipulan una fecha fija. El evento sucede cuando dicha fecha suceda. Temporal Relativo*:* Evento de tiempo que estipula la ocurrencia de un lapso de tiempo con respecto a la ocurrencia de otro evento.

- Los eventos explícito*s (*definidos por el usuario*)* son esos eventos que son detectados junto con sus parámetros por programas de aplicación (es decir, fuera del DBMS) y manejados solamente por el DBMS. Colocados una vez con el sistema, pueden ser utilizados como eventos primitivos.

1.3.2 Condiciones

Una condición puede ser un predicado sobre los parámetros del evento que disparó la regla, puede ser una consulta sobre la base de datos (si retorna filas la condición se cumple), o el llamado a un procedimiento o método que debe retornar verdadero o falso o una combinación de los anteriores. La condición no debe modificar la base de datos ni causar efectos colaterales.

1.3.3 Acciones

La acción puede realizar consultas o modificaciones sobre la base de datos, cancelar transacciones, o llamar uno o más procedimientos o métodos arbitrarios. Debido a que la acción puede realizar modificaciones sobre la base de datos, ésta puede ocasionar la ocurrencia de nuevos eventos y por tanto provocar el disparo en cascada de nuevas reglas.

Las acciones se dividen en:

-Acciones Externas: Se dan cuando son especificadas por aplicaciones, por ejemplo enviar un correo electrónico (*email*), imprimir una orden.

- Acciones Internas

Son acciones de la base de datos, como un *insert, update, select.*

1.3.4 Reglas ECA

El mecanismo utilizado con más frecuencia para dar capacidad activa a los DBMS es el uso de las reglas ECA. La E representa los eventos, estos eventos pueden ser primitivos o compuestos. La C representa la condición, cuando ocurre un evento se evalúa una condición, para saber si es verdadera o falsa. La A representa la acción, de acuerdo con el valor que devuelve la condición se ejecuta una acción que puede o no afectar a la base de datos.

Las reglas ECA se dividen en:

CA (Condición-acción): llamadas reglas de producción.
EA (Evento-Acción): llamados disparadores (*triggers*).

1.3.5 Base de datos

Una base de datos esta constituida por cierto conjunto de datos persistentes utilizado por los sistemas de aplicaciones de una empresa determinada.

1.3.6 Sistema de Administración de Base de Datos Activa

Un sistema administrador de bases de datos (DBMS) es un producto de software que provee facilidades para almacenar y administrar grandes volúmenes de información permitiendo a múltiples usuarios concurrentes acceder a tal información de una manera eficiente. Estos DBMS convencionales son considerados en la actualidad como pasivos, esto es, su funcionalidad permite crear, consultar, modificar y borrar datos como respuesta a peticiones directas a través de lenguajes declarativos o mediante lenguajes procedimentales.

En los últimos años la comunidad de investigadores en bases de datos han propuesto extender las bases de datos convencionales con nuevas capacidades para que provean mayor funcionalidad y un mejor ambiente para implementar nuevas aplicaciones.

Una de dichas extensiones consiste en transformarlas en Bases de Datos Activas, esto es, un DBMS que por sí mismo realiza operaciones automáticamente como respuesta a eventos internos (operaciones que modifican la base de datos) o externos (producidos por sensores que reportan lo sucedido en el exterior) y a condiciones que lleguen a ser satisfechas.

1.4 Reglas de las Bases de Datos Activas

El concepto sobre el cual están fundamentadas las Bases de Datos Activas es el de reglas activas, sentencias que determinan la manera como debe reaccionar la base de datos frente a situaciones expresadas en términos de eventos y/o condiciones. Las reglas en Sistemas de Bases de Datos Activas son definidas por los usuarios, o en aplicaciones o por los administradores de la base de datos. Las reglas forman la base del comportamiento para el sistema. En general las reglas están formadas por: eventos, condiciones y acciones.

Una vez que un conjunto de reglas se haya definido, el Sistema de Base de Datos Activas vigila los acontecimientos relevantes, si ocurre cualquier acontecimiento de las reglas entonces el sistema evalúa las condiciones de las reglas si resulta positivo la evaluación entonces ejecuta la acción de las reglas.

Las reglas son la base de los Sistemas de Administración de Bases de Datos Activas y aparecen bajo muchos nombres: reglas de evento-condición-acción (ECA), disparadores, monitores y alertas. El comportamiento que un sistema activo es capaz de producir se relaciona directamente con qué se puede especificar en el lenguaje de la regla, que gobierna el sistema. Un sistema de base de datos simple no implementará todas las características que son posibles usar aprovechando un sistema activo. Las reglas son controladas en asociación con algunos aspectos de la base de datos. Hay cuatro áreas donde esto es posible.

1. Una regla se puede accionar por la modificación de los datos, cuando se crea, se elimina o se modifica un objeto o se llama un método del objeto que realiza una modificación del objeto.

2. Una regla puede ser accionada cuando los datos se extraen de un objeto cuando se llama un método que extraerá datos de un objeto.

3. La base de datos puede utilizar eventos temporales como un método para accionar las reglas, donde en un intervalo de tiempo o en un momento dado se invoca una regla.

4. Una regla puede también ser accionada desde una aplicación donde se define un evento tal como una conexión del usuario y siempre que ocurra ésta conexión se dispara una regla. Los disparadores de la aplicación son de gran alcance, pues no necesitan tener acceso a la base de datos para realizar su tarea.

Las reglas activas proporcionan un nuevo método importante para diseñar las bases de datos y el tema está siendo considerado por una gran cantidad de compañías comerciales de las bases de datos.

1.4.1 Lenguaje de la regla activa

El lenguaje de la regla prescribe que se puede especificar en cada regla activa de la base de datos, la semántica de la ejecución de la regla prescribe cómo el sistema de base de datos se comporta una vez definido un conjunto de reglas.

1.4.2 Sintaxis y semántica del lenguaje de la regla activa

Brevemente, un ADBMS es un DBMS extendido con características que le permiten especificar e implementar comportamiento reactivo. Este comportamiento reactivo se especifica utilizando el paradigma de reglas activas, o reglas ECA (Evento, Condición, Acción). Las reglas activas presentan una sintaxis similar a la siguiente:

DEFINE RULE <Nombre de Regla> *ON* <Evento>
IF <Condición> *DO* <Acción>

La semántica de una regla activa es: "Cuando ocurra el Evento, se evalúa la Condición y si ésta se cumple, es decir es verdadera, se ejecuta la Acción".

1.4.3 Aspectos teóricos de las reglas activas

Aunque muchos sistemas pueden utilizar un lenguaje de regla activa similar, la manera en que las reglas se interpretan puede variar extensamente. Pero se tiene que contar con la detección de evento, el método de la ejecución de la regla, la resolución del conflicto, y el modo de acoplamiento.

1.4.3.1 Detección de eventos

Una vez que las reglas han sido definidas, el ADBMS comienza a observar la ocurrencia de eventos lo cual se denomina detección de eventos, y aquellos que sean de interés para las reglas definidas son utilizados para implementar la semántica de dichas reglas.

1.4.3.2 Ejecución de la regla

Cuando ocurre el evento de una regla, se dice que la regla se dispara y entonces se debe hacer la ejecución de la regla, es decir, evaluar la condición si procede. La ejecución de la regla se basa en el modelo de ejecución.

1.4.3.3 Modelo de ejecución de reglas

Un ADBMS debe ofrecer un lenguaje para la especificación de eventos y reglas activas. Adicional a la especificación de reglas y la detección de eventos, una debe ofrecer un modelo de ejecución de reglas.

Un modelo de ejecución determina cuándo son ejecutadas las reglas y que propiedades acompañan a dicha ejecución. En general los eventos suceden dentro de transacciones y las reglas se ejecutan dentro de transacciones también. La transacción que ocasionó el evento que disparó la regla se denomina transacción del evento, mientras que la transacción donde se ejecuta la regla se denomina transacción de la regla.

Un modelo de ejecución puede determinar por ejemplo, que la transacción de la regla debe ser distinta a la transacción del evento. También puede especificar una relación de dependencia entre dichas transacciones, por ejemplo que si la transacción del evento aborta, entonces la transacción de la regla también debe ser abortada.

1.4.3.4 Modo de acoplamiento

Una segunda relación qué puede especificar un modelo de ejecución, se denomina modo de acoplamiento, que permite estipular cuándo debe ser iniciada la transacción de la regla con respecto a la del evento. Se pueden dar los siguientes modos de acoplamiento:

- Acoplamiento inmediato: Acoplamiento inmediato significa que la transacción de la regla es ejecutada inmediatamente después que el evento de la regla es detectado.

- Acoplamiento diferido: Acoplamiento diferido significa que la transacción de la regla es ejecutada al final de la transacción del evento justo antes de hacer commit.

- Acoplamiento desacoplado: significa que la transacción de la regla es iniciada como una transacción separada, es decir, en una transacción nueva.

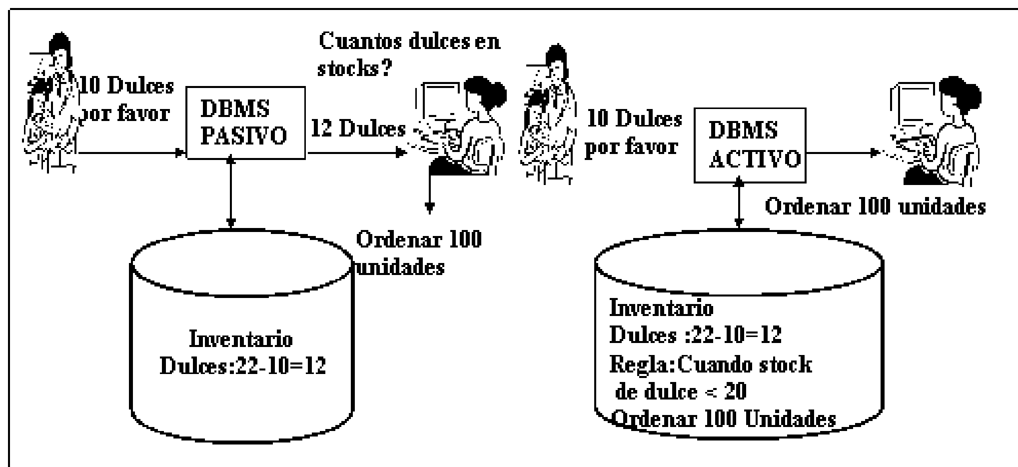
1.4.3.5 Resolución de conflicto

Cuando dos o más reglas se disparan en un mismo punto en el tiempo se dice que existe un conjunto de reglas en conflicto. Debido a que distintas órdenes de ejecución de reglas pueden ocasionar estados distintos en la base de datos, el ADBMS debe ofrecer un mecanismo de resolución de conflictos. Un mecanismo puede ser el uso de prioridades en reglas, según el cual, la prioridad indica el orden en que deben ser ejecutadas. La resolución de conflictos es parte del modelo de ejecución de reglas.

1.5 Bases de Datos Activas vrs. Bases de datos pasivas

Los DBMS tradicionales son considerados en la actualidad como pasivos, ya que su funcionalidad permite crear, consultar, modificar y borrar datos sólo como respuesta a peticiones directas emitidos por usuarios o programas de la aplicación. En la figura 2 se muestra la diferencia entre una y una base de datos pasiva.

Figura 2. DBMS activos vrs. DBMS pasivo



Los Sistemas de Bases de Datos Activas son significativamente más poderosos que sus contrapartes los sistemas de bases de datos pasivos:

- Los Sistemas de Bases de Datos Activas pueden ejecutar eficazmente funciones, esto en sistemas de bases de datos pasivos debe codificarse en las aplicaciones.

- Los Sistemas de Bases de Datos Activas sugieren y facilitan aplicaciones más allá del alcance de sistemas de bases de datos pasivos.

- Los Sistemas de Bases de Datos Activas pueden realizar tareas que requieren subsistemas de propósito especial en sistema de base de datos pasivo.

Ejemplos de tareas que pueden ser realizadas por Sistemas de Bases de Datos Activas, pero que en sistemas de bases de datos pasivas se requiere subsistemas de propósito especial son: restricción de integridad, permisos, y vistas.

La base de datos activa, a veces se refiere a alertas o a monitores, especifica ciertas acciones que deben ser invocadas siempre que se detecte ciertas condiciones. Por ejemplo, un disparador de control de inventario podría detectar cuando el nivel es bajo y automáticamente debe ejecutar una acción para realizar un nuevo pedido de artículos, como se muestra en la figura 2.

Ejemplos de aplicaciones que se sugieren o son facilitadas por Sistemas de Bases de Datos Activas, pero están más allá del alcance de sistemas de bases de datos pasivos, son sistemas expertos y manejadores de procesos de negocio *(workflow)*.

2. SINTAXIS DEL LENGUAJE DE CDOL BASADO EN REGLAS

2.1 ¿Qué es *CDOL*?

CDOL (por sus siglas en ingles *Comprehensive, Declarative Object Language*). Es un lenguaje basado en las reglas de Bases de Datos Activas, conocido como lenguaje de objeto comprensivo, declarativo que es una integración de la tecnología deductiva, orientada a objetos, y de Bases de Datos Activas. *CDOL* proporciona un sublenguaje para la expresión de datos derivados, restricciones, actualizaciones y reglas activas.

El lenguaje de consulta basado en las reglas de *CDOL* proporciona un acercamiento expresivo para extender las bases de datos, almacenar atributos y clases. El sublenguaje de restricción permite la especificación de las restricciones para la integridad de la base de datos para que mantenga un estado coherente.

El sublenguaje de actualización de *CDOL* permite las peticiones declarativas de actualización, donde las actualizaciones están encapsuladas en los métodos asociados a las definiciones de clase, así integrándose con los conceptos de diseño orientado a objetos tradicionales.

Las reglas activas en general se utilizan para vigilar la ocurrencia de eventos específicos y servir como alertas y disparadores dentro de una aplicación de *CDOL*.

2.2 Símbolos básicos

El significado de los símbolos usados en la sintaxis para definir la gramática de *CDOL*, se describe en la tabla II.

Tabla II. Símbolos de la gramática de CDOL

<table>
<tr><th>SÍMBOLO</th><th>DESCRIPCIÓN</th></tr>
<tr><td>:=</td><td>Se utiliza para indicar una asignación</td></tr>
<tr><td>|</td><td>Se utiliza para indicar alternativa</td></tr>
<tr><td><></td><td>Para representar simbolos no terminales</td></tr>
<tr><td>{}</td><td>Para representar construciones opcionales, cero o mas</td></tr>
<tr><td>[]</td><td>Para representar construciones opcionales, mas de una</td></tr>
<tr><td>…</td><td>Denota continuación</td></tr>
<tr><td>‘ ‘</td><td>Para representar simbolos terminalaes</td></tr>
</table>

Las producciones más elementales son:

<digit> ::= 0|1|..|9
<lowercaseletter>::= a|...|z
<uppercaseletter>::= A|...|Z
<comparisonsymbol>::= =| < | > | <= | >= | <>
<assignmentsymbol> ::= :=
<sign>:= +|-
<setmembershipsymbol>:= sub|prosub | in | =
<set_operator>:= union|dif|int |+ | -
<specialsumbol>:= _| + | - |) |(|*| $ | % | &| # | @| !
<letter>:= <uppercaseletter> | <lowercaseletter>
<letterordigit>:= <letter>| <digit>
<booleavalue>:= true | false
<unsignedinteger>:= <digit>| {<digit>}
<integer>::= [<sign>]<unsignedinteger>
<realnumber>::= [<integer>].<unsignedinteger> | [(E|e)<sign><unsignedinteger>]
<numeric>::= <integer>| <realnumber>
<character>::= '<letterordigit>'|<'<specialsymbol>'>| ''
<lowercasestring>::= <lowercaseletter>{(<letterordigit>|)}
<upperrcasestring>::= <uppercaseletter>{(<letterordigit>|)}
<set> ::= <setconstant>| 'f' <variable > 'g'
<setconstant> ::= 'f'<constant>{, <constant > 'g'}|'f' 'g'
<string> :::= "<character>{<character>}"
<constant>::= <numeric>| <string>|<character>| <setconstant>

2.3 Tipos

Los tipos se pueden clasificar en tipos simples y tipos del constructor. Los tipos simples se clasifican en los tipos del modelo y los definidos por el usuario. Los tipos del constructor se clasifican en los tipos de estructuras y los tipos enumerados. Entre los tipos se incluyen caracter (*character*), falso o verdadero (*boolean*), punto flotante, y enteros. Otros tipos que se utilizan en *CDOL* son los conjuntos, las listas, las matrices y las cadenas.

```
<type_spec> ::= <simple_type_spec> | <constr_type_spec>
<simple_type_spec>::=<base_type_spec><template_type_spec><scoped_type_name>
<constr_type_spec> ::= <struct_type> | <enum_type>
<base_type_spec> ::= <char_type>|<boolean_type| <floating_pt_type>|<integer_type>
<char_type> ::= char
<boolean_type> ::= boolean
<floating_pt_type> ::= float | double
<integer_type> ::= <signed_int> | <unsigned_int>
<signed_int> ::= <signed_long_int> | <signed_short_int>
<signed_long_int> ::= long
<signed_short_int>::= short
<unsigned_int>::= <unsigned_long_int> | <unsigned_short_int>
<unsigned_long_int> ::= unsigned long
<unsigmed_short_int>::= unsigned short
<template_type_spec>::= <set_type> | <list_type> | <bag_type> | <array_type> | <string_type>
<set_type>::= set '<' <type_spec> '>'
<list_type>::= list '<' <type_spec> '>'
<bag_type>::= bag '<' <type_spec> '>'
<array_type>::= <array_spec>'<' <type_spec>,<positive_int_const> '>'| <array_spec>'<' <type_spec> '>'
<array_spec>::= array
<positive_int_const>::= [+] <unsignedinteger>
<string_type>::= string '<' <positive_int_const> '>' | string
```

La estructura y los tipos enumerados son similares a los del lenguaje de C++.

```
<struct_type>::= struct < struct_tag> 'f' <menber_list > 'g'
<struct_tag>::= <identifier>
<member_list>::= <member> | <member><member_list>
<member>::= <type_spec><struct_member_label>;
<struct_member_label>::= <identifier>
<enum_type>::= enum <enum_tag> 'f' <enumerator_list> 'g'
<enum_tag>::= <identifier>
<enumerator_list>::= <enumerator> | <enumerator>,<enumerator_list>
<enumerator>::= <identifier>
```

2.4 Construcciones básicas del modelo

En sistemas orientados a objetos, los objetos se clasifican en clases para capturar y encapsular características estructurales y de comportamientos comunes. En *CDOL*, los objetos se modelan en clases. Cada clase es identificada por un nombre de clase que es un identificador. Un identificador en *CDOL* es una cadena que comienza con una letra minúscula y conteniendo una secuencia de letras, dígitos, y el carácter de subrayado '_'.

```
<class_name>::= <identifier>
<identifier>::= <lowercasestring>
<scoped_type_name>::= <class_name>::<scoped_type_name>|<class_name>|<type_name>
<type_name>::= <identifier>
```

Las características estructurales se modelan como propiedades. Las propiedades pueden ser clasificadas como atributos y relaciones. Dentro de una clase, las propiedades son identificadas por los nombres de los atributos y los nombres de las relaciones. Las características del comportamiento se modelan como métodos.

```
<property>::= <attribute_name> <relationship_name>
<attribute_name>::= <identifier>
<relationship_name>::= <identifier>
<method_name>::= <identifier>| ˜<identifier>
<scoped_method_name>::= <scoped_type_name>::<method_name>
```

Las reglas pueden ser agrupadas en conjuntos de regla que forman las agrupaciones lógicas con el fin de referenciar fácilmente a una gran cantidad de reglas.

```
<virtual_param_class>::= <virtual_class_name> ['('<arg_list>')']
<virtual_class_name>::= <identifier >
<virtual_param_attr>::= <virtual_attr_name> ['('<arg_list>')']
<virtual_attr_name>::= <identifier >
<rule_name>::= <identifier >
<rule_set_name>::= <identifier >
<arg_list>>::= <arg> | <arg> , <arg_list>
<arg>::= <type_spec>| <variable>
```

Además de concepto de clase, *CDOL* también utiliza el concepto de relación. Las relaciones en *CDOL* son de mayor alcance que las relaciones en los DBMS relacionales porque permite que los objetos sean almacenados con sus atributos y sus relaciones.

<relation_name>::= <identifier >

CDOL utiliza transacciones como otros procedimientos de almacenamiento, además de los métodos de la clase. La diferencia principal entre un método y una transacción es que el método está asociado a una clase específica pero no es una transacción.

<transaction_name>::= <identifier >
<function_name>::= <identifier >

CDOL utiliza comportamientos activos en las bases de datos con reglas activas que pueden ser agrupadas en conjuntos de reglas activas que forman las agrupaciones lógicas para ser referenciadas de una forma más fácil. *CDOL* también utiliza la definición explícita de las restricciones para la integridad de las aplicaciones.

<active_rule_name>::= <identifier >
<active_rule_set_name>::= <identifier >
<constraint_name>::= <identifier >

2.5 Expresiones

Una expresión puede evaluar cierto valor u objeto. La constante *NIL* es una expresión que significa indefinido. Hay 5 unidades básicas de expresión, que son variables, notaciones de punto, llamadas de función, llamadas de transacción, y funciones agregadas.

<expression>::= <value_expr > | <obj_expr | nil
<variable>::= <uppercasestring >
<dotnotation>::= <variable > <path_component>{.< path_component >}
<path_component>::=<attribute_name>|<relationship_name>|<method_call>|<virtual_attr_name>
['('<actual_arg_list>')'] |<struct_member_label>
<method_call>::= <method_name>'('['<actual_arg_list>']')'
<actual_arg_list>::= <expression>|<expression>[,<actual_arg_list>]

Las funciones son llamadas por sus nombres junto con cualquier argumento que las funciones esperen. Similar a las funciones, las transacciones son llamadas por sus nombres junto con cualquier argumento que las transacciones esperen.

```
<func_call>::=<function_name>'('[<actual_arg_list>]')'
<transaction_call>::=<transaciton_name>'('[<actual_arg_list>].')'
```

Las funciones agregadas pueden realizar las siguientes operaciones matemáticas, promedio (*average*), suma (*sum*), producto (*product*), número de elementos, el mínimo y el máximo de un conjunto de elementos incluidos en las propiedades. El agregado *COUNT* (contar) se puede aplicar a cualquier atributo de la colección o las relaciones para producir un resultado numérico. El promedio, la suma, y los agregados del producto se pueden aplicar a cualquier atributo numérico para producir un resultado numérico. Los agregados del mínimo y del máximo se pueden aplicar a cualquier atributo simple o a la colección de atributos de tipo entero, cadena, carácter.

```
<aggregate_func>::=<aggegate_sumbol>'('[<aggregate_body>]')'
<aggegate_sumbol>::=avg | sum | product |count | min| max
<aggregate_body>::=<mv_aggregate_path>| 'f'<mv_aggregate_path> 'g' |
<sv_aggregate_path>over<aggregate_binding>|
'f'<sv_aggregate_path> 'g' over<aggregate_binding>
<mv_aggregate_path>::=<mv_variable>|<sv_variable>[<sv_path>]<mv_end>
<sv_path>::= <path_component>
<mv_end>::= <path_component>
<sv_aggregate_path>::=<sv_variable>[.<sv_path>]
<aggregate_binding>::=<rule_body>
<mv_variable>::= <variable>
<sv_variable>::= <variable>
```

Un valor de expresión es el resultado de evaluar una expresión aritmética o *boolean* o caracter.

```
<value_expr>::= <bool_expr>|<arith_expr>|<char_expr>
<bool_expr>::=<bool_expr> or <bool_term>|<bool_expr> xor <bool_term>| <bool_term>
<bool_term>::=<bool_term>and<bool_factor>|<bool_factor>
<bool_factor>::=[not]<bool_primary>
<bool_primary>::=<booleanvalue>|'('<bool_expr>')'|<comparisonatom>          |<setmembershipatom>
|<dotnotation>| <func_call>|<transaction_call>   |<variable>|<aggregate_func>
<arith_expr>::= <arith_expr> + <arith_term>|<arith_expr> <arith_term>|<arith_term>|
```

<arith_term>::= <arith_term> * <arith_factor>|<arith_term> / <arith_factor> |
<arith_term>%<arith_factor>|<arith_factor>
<arith_factor>::=<arith_primary> ^ <arith_factor>|<arith_primary>
<arith_primary>::=<numeric>|<variable>|<dotnotation>|func_call>|<transaction_call> |<aggregate_func> |'(' <arith_expr> ')'
<char_expr>::=<character>|<variable>|<dotnotation>|<func_call>| <transaction_call>|<aggregate_func>
<obj_expr>::=<variable>|<dotnotation>| <func_call>|<transaction_call> |<set_expr>|<string_expr>
<set_expr>::=<set_expr><setoperator><set_facor>|<set_factor>
<set_factor>::= <set> | <variable> | <dotnotation> | <func_call>| <transaction_call>| '(' <set_expr> ')
<string_expr>::= <string> | <variable> | <dotnotation> | <func_call>| transaction_call>| <aggregate_func>

2.6 Lenguaje de consulta de la regla

El lenguaje de consulta de la regla de *CDOL* es utilizado actualmente para derivar clases virtuales, atributos virtuales, y relaciones. Una regla de consulta consiste en un encabezado de la regla y un cuerpo de la regla. El encabezado de la regla especifica que se está derivando y el cuerpo de la regla proporciona a una especificación declarativa de la derivación.

<rule>::= <rulehead>'<-' <rulebody>
<rulehead>::= <virtual_class_head>|<virtual_attr_head> |<relation_head>
<virtual_class_head>::= <virtual_param_class> : <variable>
<virtual_attr_head>::= (<class_name>|<virtual_class_name>) : <variable> '[' <virtual_attr_binding>']'
<virtual_attr_binding>::= <virtual_attr_name> ['(' <arg_del_list>')']- <expression>
<relation_head>::= <relation_name>'[' <prop_binding>{,<prop_binding>}']'
<prop_binding>::= <dot_component>= <expression>
<dot_component>::= <path_component>{.<path_component>}

El cuerpo de una regla es una lista de literales con la semántica conjuntiva entre ellos. Los literales pueden ser positivos o negativos. Los literales positivos se pueden cuantificar o no cuantificar. Finalmente, no cuantificar literales son las formas más primitivas de literales, también se conocen como átomos.

<rulebody>::= <literal>{,<literal>}
<literal>::= <positive_literal>| <negative_literal>
<positive_literal>::= <quantified_literal>| <unquantified_literal>
<negative_literal>::= not<positive_literal>not'('<literal>{,<literal>}')'
<quantified_literal>::=<quantification_spec>'('<literal>{,<literal>}')'
<quantification_spec>::= <quantifier><quantification_var>
<quantifier>::= exists | for_all

<quantification_var>::= <variable> in <set_expr>| [,<quantification_var>]
<unquantified_literal>::= <atom>
<atom>::=<comparisonatom> | <setmembershipatom> | <term>
<comparisonatom>::=<expression><comparisonasymbol><expression>
<setmembershipatom>::=<set_expr><setmembershipsymboll><set_expr>
<term>::=<ide_term><objecterm> | <relarionterm>| <parameterizedterm>

Un identificador de término une una variable a las instancias de una clase. Un término del objeto une una o más expresiones a una o más propiedades de los objetos limitados por una variable. Un término de la relación une una o más expresiones a uno o más atributos de una relación. Un término con parámetros une una variable a las instancias de una clase con parámetros virtuales.

<ide_term>::= (<class_name> | < virtual_class_name >):<variable>
<objecterm>::=<id_term>'[<prop_binding>{,<prop_binding>}']'|<variable>'['<prop_binding>
{,<prop_binding>}']'
<relationterm>::= <relation_term >'[' <prop_binding>{,<prop_binding>}']'
<parameterizedterm>::= <virtual_class_name> ['(' < actual_arg_list >')']:<variable>

2.7 Sublenguaje de restricción

Una restricción contiene una presunción y una implicación. Si la presunción está satisfecha, entonces se da el caso que la implicación también está satisfecha. La presunción y la implicación pueden tomar la forma de un cuerpo de la regla. Además, la implicación puede ser una constante booleana que toma el valor *FALSE*. Si la implicación de una restricción es una constante booleana de valor *FALSE*, la restricción es llamada una negación, indicando que la presunción no se puede satisfacer en ningún estado de la base de datos.

<constraint>::= if <presumption> then <implication>
<presumption>::= <rulebody>
<implication>::= <rulebody>|false

2.8 Sublenguaje de actualización

Una regla de actualización consiste en un encabezado de actualización y un cuerpo. El encabezado de la regla especifica la actualización que pueda realizarse con las posibles actualizaciones de los objetos así como las actualizaciones a las propiedades. El cuerpo de la regla especifica las actualizaciones del objeto. Las actualizaciones del objeto incluyen creaciones del objeto, destrucción del objeto, y migraciones de objetos. Las actualizaciones de las propiedades y la creación o la migración de objeto se pueden especificar al mismo tiempo en una regla.

```
<update_rule>::= <update_rule_head>['<-' <update_rule_body>];
<update_rule_head>::= <object_update_head>| <property_update_head> | <generic_update_head>
<update_rule_body>::= <rule_body >
<object_update_head>::=class_name>:(<create_object>|<destroy_object>)| <migrate_object>
<create_object>::= new <variable> ['[' <prop_update> {, <prop_update>}']']
<destroy_object>::= destroy <variable>
<migrate_object>::= migrate <variable> ['[' <prop_update> {, <prop_update>}']']
```

Las actualizaciones de las propiedades incluyen la actualización a los atributos o a las relaciones de los objetos. Cuando las propiedades son actualizadas solo se actualiza el valor de la propiedad.

```
<property_update_head>::= <id_term> ['[' <prop_update> {, <prop_update>}']']
<prop_update>::= <update_single_valued_prop> | <update_multivalued_prop>
 <update_single_valued_prop> ::= ( <attribute_name> | <relationship_name> )
 <assignmentsymbol> <expression>| <method_name>  '(' <expression> ')'
<update_multivalued_prop>::= <updaate_one_element>| <update_a_set>
<update_one_element> ::= ( ( <attribute_name> | <relationship_name> )
                    <assignmentsymbol> 'f' <sign> <expression>'g' ) <method_name>'(' <expression> ')'
<update_a_set> ::= ( ( <attribute_name> | <relationship_name> ) <assignmentsymbol>  <set_expr> ) |
                   <method_name>'(' <set_expr> ')'
```

En vez de que los objetos sean manipulados directamente, las actualizaciones genéricas alcanzan actualizaciones con métodos o transacciones. Este mecanismo utiliza encapsulamiento y permite actualizaciones a las propiedades de un objeto fuera de la clase del objeto de una manera controlada.

```
<generic_update_head>::= <id_term>'[' <method_call>']' | <transaction_call>
```

2.9 Lenguaje de la regla activa

Las reglas activas en *CDOL* pueden ser reglas de Evento-Condición-Acción (*ECA*), reglas de Evento-Acción (*EA*), y reglas de Condición-Acción (*CA*). Las reglas activas utilizan opciones de prioridad para controlar la ejecución de las reglas accionadas por los mismos eventos. Accionar un evento en *CDOL* es llamar a un método y a una transacción, con la opción para especificar el momento inmediatamente antes o después de que se invoque el método o la transacción. Los eventos múltiples se pueden especificar en una regla activa con la semántica disyuntiva entre ellos. Cualquier variable del argumento de la lista de variables de un evento puede ser referenciada en la condición o la acción de la misma regla activa.

```
<active_rule> ::= (<ec_part>|<e_part>|<c_part>)<a_part><priority_option>
<ec_part>::= <e_part><c_part>
<e_part>::= event<triggerin_event_list>
<triggerin_event_list>::= [before|after]<triggering_event>[or<triggering_event_list>]
<triggerin_event>::= <scoped_type_name>: <variable>'['<method_name>
                     '(' [arg_var_list]')' ']' | <transaction_name>'(' [arg_var_list]')'
<arg_var_list>::= <arg_var><arg_var>, <arg_var_list>
<arg_var>::= <variable>
```

Las condiciones en *CDOL* utilizan modos de ejecución que son inmediatos, diferidos. Si no se especifica ningún modo de acoplamiento, el modo inmediato de acoplamiento se toma como valor por defecto. Una condición es una consulta expresada en la forma del cuerpo de la regla.

```
<c_part>::=condition [<condition_coupling>]<condition><condition_coupling>::=<coupling_mode>
<coupling_mode>::= inmediate | deferred | decoupled
<condition>::= <rule_body>
```

Las acciones en *CDOL* utilizan modos inmediatos, diferidos, y desacoplados opcionales con respecto a eventos en reglas de EA y con respecto a la evaluación de la condición en reglas de ECA o de CA. Si no se especifica ningún modo del acoplador, el modo inmediato del acoplamiento se toma como valor por defecto. Una acción es una secuencia de las reglas de actualización. La opción de prioridad permite definir el orden en que serán ejecutadas las reglas con respecto a un conjunto de ellas.

<a_part>::= action[<action_coupling>]<action_list>
<action_coupling>::=<coupling_mode>
<action_lisst>::= <action >[<action_list>]
<action>::= <update_rule>
<priority_option>::= ['>' <before_Active_rule_list>]['<'<after_active_rule_list>]
<before_Active_rule_list>::= <active_rule_list>
<after_Active_rule_list>::= <active_rule_list>
<active_rule_list>::= (<active_rule_name>| <active_rule_set_name>) {,<active_rule_list>}

2.10 Idioma de definición de datos

El Lenguaje de definición de datos (*LDD*) de *CDOL* utiliza la definición de los esquemas de la base de datos. El tipo, la clase, el método, la clase virtual, la función, la regla, el conjunto de la regla, y las declaraciones de la regla activa se pueden definir usando el *LDD*.

<specofication> ::= <definition>; | <definition>;<specofication>
<definition>:: <type_def>| <class_del>|<method_del>| <transaction_del> <virtual_class_del>|
<func_call><constraint_id>| <rule_del> <rule_act_del> <active_rule_del>

La definición de tipos hace más fáciles definir tipos de usuario. Las declaraciones de la clase definen la estructura de clases. La cabecera de la clase especifica el nombre de la clase, las especificaciones de la herencia. El cuerpo de la clase declara tipos, atributos, relaciones, atributos virtuales, y los prototipos del método para la clase.

<type_del>::= typedef <type_spec><type_name>
<persistence_id>::= persisten | transient
<class_header> ::= class <class_name > [: <inheritance_spec>][type_property_list]
<inheritance_spec>::= <scoped_type_name> [,<inheritance_spec>]

```
<type_property_list>::= '('[<extent_spec>] [key_spec]')'
<extent_spec>::= extent <extent_name>
<key_spec>::= key[s] <key_list>
<key_list>::= <key> <key> , <key_list>
<key>::= <property>| '(' <property_list>')'
<property_list>::= <property> | <property>,<property_list>
<class_body>::= <export>| <export><class_body>
<export>::= <type_del>;| <attr_del>; | <rel_del>; | <virtual_attr_del>;| <method_proto_del>;
<attr_del>::= [readonly]attribute <type_spec><attribute_name>
<rel_del>::= relationship <target_of_path> <relarionship_name>|[inverse <inverse_traversal_path>]
<target_of_path>::= <class_name>| <rel_collection_type> '<' <class_name> '>'
<inverse_traversal_path> ::= <class_name>:: <relationship_name>
<rel_collection_type>::= set | list | bag array
<virtual_attr_del> ::= virtual <type_spec> <virtual_param_attr> = <rule_name>
<method_proto_de;>::= <return_type> <method_name> '(' [<arg_del_list>] ')'
<arg_del_list>::= <arg_del>| <arg_del>, <arg_del_list>
<arg_del>::= [<arg_moode>]<type_spec> : <variable>
<arg_mode> ::= in | out | inout
<method_del>::= method  <return_type><scoped_method_name> '(' [<arg_del_list>] ')'   'f' {<update_rule>}
'g'
<return_type>::= <type_spec> | void
```

Las transacciones en *CDOL* son similares a los métodos excepto que no están asociadas a ninguna clase. Cada declaración de la transacción especifica el tipo que retorna, el nombre, los argumentos, y la puesta en práctica de la transacción como secuencia de las reglas de actualización.

```
<transaciton_del>::=transaction<return_type><transaction_name>'('[<arg_del_list>]')''f'{<update_rule>  }   'g'
```

Antes de que una función externa pueda ser utilizada, el sistema debe saber el nombre, el tipo que retorna, el número y el tipo de cada argumento. Las declaraciones de restricciones son definiciones específicas de restricciones.

```
<func_del> ::= function <return_type> <function_name> '(' [<arg_del_list>]')'
<constraint_del> ::= constraint <contraint_name> <constraint>
```

Las reglas activas se pueden poner opcionalmente en conjuntos de la regla.

```
<rule_del> ::= rule <rule_name> [in  <rule_set_name>] 'f' {<rule> } 'g'
<rule_set_del> ::= ruleset <rule_set_name>
<active_rule_del> ::= active <active_rule_name> [in  <active_rule_set_name>] 'f' <active_rule>  'g'
```

3. SISTEMAS DE ADMINISTRACIÓN DE BASE DE DATOS ACTIVAS

3.1 Introducción

Anteriormente, se ha definido que una almacena la semántica de los datos además de los propios datos. En este capítulo, se describen como los diferentes Sistemas de Administración de Base de Datos Activa, tanto relacionales, como orientados a objetos y relacionales orientados a objetos, manejan esta semántica, con el uso de disparadores y reglas activas.

3.2 Sistemas de Base de Datos Activas Relacionales

3.2.1 *Orácle*

Orácle es un sistema de administración de bases de datos relacional. En *Orácle*, las principales estructuras de almacenamiento son tablas e índices. Cada usuario creado en la base de datos le es asignado un esquema dentro del cual puede crear sus propias estructuras de almacenamiento, programas, vistas, etc. Un usuario puede otorgarles permisos a otros usuarios de la base de datos para que accedan a sus objetos (modificar, consultar tablas y/o ejecutar programas).

Un programa en *Orácle* está constituido por procedimientos almacenados escritos en lenguaje *PL/SQL*, un lenguaje procedimental que permite embeber sentencias *SQL*. Con el fin de poder administrar y organizar mejor los programas, *Orácle* permite agrupar varios procedimientos y funciones en uno o más paquetes. Para poder ejecutar un programa dentro de un paquete se utiliza la notación: paquete.procedimiento.

Sí un usuario desea ejecutar un procedimiento de otro usuario debe utilizar la notación: usuario.paquete.procedimiento. Las aplicaciones son interfaces gráficas que hacen llamados a procedimientos almacenados para desarrollar su trabajo.

3.2.1.1 *Roke*

Roke (por sus siglas en ingles: *Rule Oriented worK Environment*) es un ambiente para la definición y ejecución de reglas activas construido sobre *Orácle*. La arquitectura de *Roke* está ligeramente acoplada con el motor de bases de datos subyacente debido a que sus estructuras y software no fueron integrados con el software interno del motor *Orácle*.

3.2.1.2 Arquitectura *Roke*

El objetivo de *Roke* es proveer un ambiente que permita crear, manejar y ejecutar reglas activas. Roke está compuesto fundamentalmente por un lenguaje denominado *ARENA* (por sus siglas en ingles*: Active rule languaje*) el cual permite definir reglas activas y un conjunto de módulos de software encargados de suministrar propiedades atribuidas a Bases de Datos Activas como definición y manejo de reglas, detección de eventos y disparo de reglas. El objetivo principal de *Arena* es ofrecer un lenguaje que permita especificar reglas activas con la sintaxis y semántica adecuadas para satisfacer los requerimientos en un dominio de aplicaciones específico como la coordinación de procesos de negocio (*workflow).* *ROKE* extiende el motor subyacente con las siguientes capacidades:

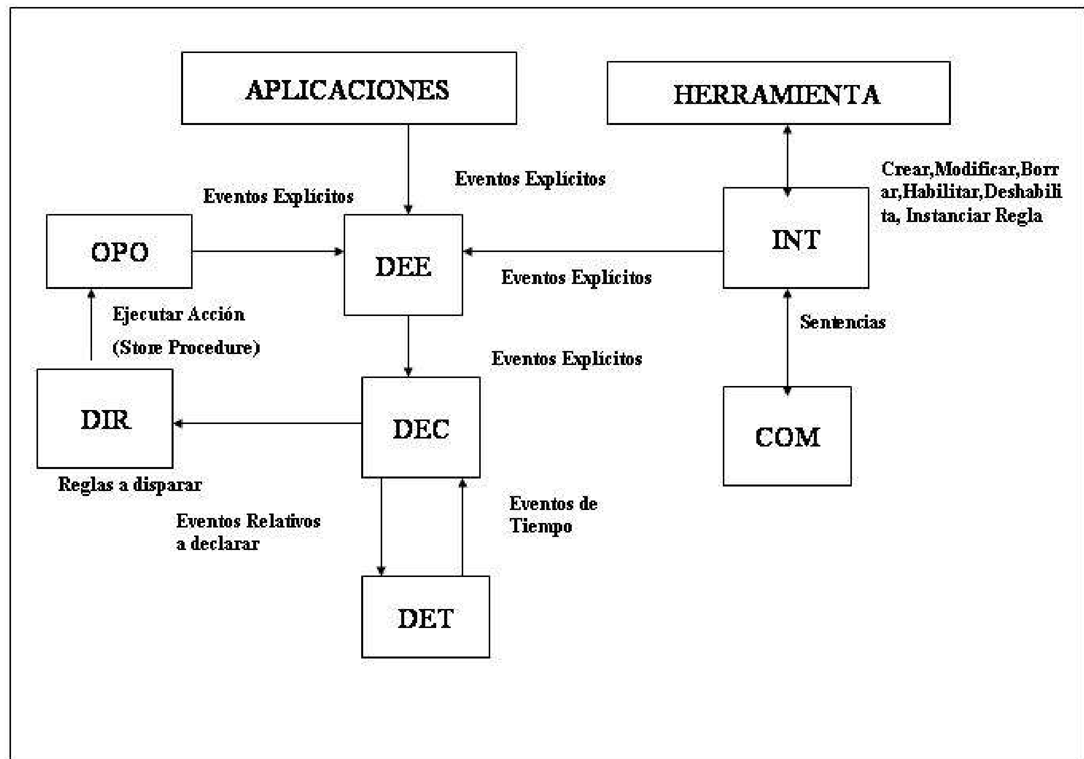
- Definición y ejecución de reglas activas
- Detección de eventos primitivos: explícitos, enmascarados y temporales.
- Detección de eventos compuestos

Roke implementa todas las estructuras de almacenamiento y programas dentro del esquema de un usuario *Orácle* denominado *Roke*.

El usuario *Roke provee* todos los permisos adecuados para que los demás usuarios creados en la base de datos puedan acceder a las capacidades ofrecidas por el ambiente *Roke* tales como crear y ejecutar reglas.

Roke utiliza tablas e índices como estructuras de almacenamiento y procedimientos almacenados escritos en *PL/SQL* para la implementación de sus componentes de software. Todas las reglas creadas por todos los usuarios en la base de datos son almacenadas dentro del esquema del usuario *Roke*. Las reglas, condiciones y acciones son tratadas como entidades independientes cada una con un identificador único; cada regla, condición y acción va acompañada del usuario que la creó, es decir, su dueño. La arquitectura que utiliza *Roke* se muestra en la figura 3.

Figura 3. Arquitectura *Roke*



Los componentes de la arquitectura son los siguientes:

- DEE (Detector de Eventos Explícitos): Es en realidad una interfaz que ofrece el procedimiento *RaiseEvent* mediante el cual se puede generar eventos explícitos y sus parámetros.

- DET (Detector de Eventos Temporales): Es un proceso que ejecuta un ciclo infinito dentro del cual realiza la detección de eventos temporales absolutos accediendo al reloj del sistema. También realiza la detección de eventos temporales relativos en coordinación con el DEC.

- DEC (Detector de Eventos Compuestos): Los eventos compuestos en *Roke* son formados mediante la combinación de eventos primitivos y/o compuestos con operadores de eventos tales como *AND*, *OR* y *NOT*. Este componente detecta eventos compuestos y las reglas interesadas en tales eventos son puestas en una cola de disparos para que el componente disparador de reglas DIR las ejecute.

- DIR (Disparador de Reglas): Realiza la ejecución de reglas, es decir, evalúa la condición de la regla y si ésta se cumple, efectúa la acción de la regla. Este componente ejecuta todas las reglas que se encuentren en una cola de disparos que va llenando el componente DEC.

- OPO (Opción Procedimental de *Orácle*): La opción procedimental de *Orácl*e provee el ambiente necesario para la creación de procedimientos almacenados y su ejecución. En *Roke*, la acción de una regla puede hacer llamados a procedimientos almacenados en la base de datos. Los procedimientos a los cuales haga referencia una regla deben existir en el esquema del usuario que la creó.

- COM (Compilador): El compilador realiza el análisis léxico y sintáctico de sentencias en el lenguaje *Arena* (*Active rule languaje*).

- INT (Interfaz): Esta interfaz provee servicios tales como crear, modificar, consultar y borrar reglas.

3.2.1.3 *Arena*

Arena es el lenguaje de regla activa que utiliza *Roke*, en Arena se utiliza la sintaxis y semántica Evento-Condición-Acción para la construcción de reglas. La sintaxis general de una regla activa en Arena es la siguiente:

CREATE [SKELETON] RULE nombre_regla *ON* Evento
IF Condición *DO* Acción

Dos tipos de reglas pueden ser definidos en *Arena:* reglas completas y reglas esqueleto. Una regla es completa si no posee variables en ninguno de sus componentes (Evento, Condición y Acción), y en caso contrario se dice que es una regla esqueleto.

A) VARIABLES

Una variable es un identificador que puede tomar cualquier valor (no tiene un tipo de dato predefinido) de cualquiera de los tipos de datos soportados en Arena: *NUMBER* y *VARCHAR*. Las variables se estipulan con un identificador precedido por el símbolo pesos ($). Una variable convierte a la regla que la contiene en un esqueleto o plantilla.

Las variables pueden ser utilizadas en:

- La máscara en un evento enmascarado (máscaras esqueleto).
- En los argumentos de los procedimientos llamados en la acción de una regla (acción esqueleto).

Ejemplo:
CREATE SKELETON RULE MiRegla *ON* compra IF compra.valor > 1.000.000
DO aplicar_descuento (compra.numero_factura, $valor_descuento);

Esta regla esqueleto estipula una regla del negocio según la cual "se debe aplicar cierto porcentaje de descuento a compras superiores a un millón de pesos". De acuerdo a las condiciones del mercado la organización estipula qué porcentaje (valor_descuento) aplica sobre montos superiores a un millón de pesos y utiliza dicho valor para instanciar la regla anterior.

B) REGLAS ESQUELETO

Se dice que una regla es una regla esqueleto si esta posee en su definición cualquiera de los siguientes componentes:

- Una máscara esqueleto
- Una acción esqueleto

Se dice que una regla esqueleto es instanciada cuando le son dados valores constantes a sus variables. Cuando una regla esqueleto es instanciada, dicha instancia se convierte en una regla completa y de hecho una nueva regla completa es creada con un nuevo identificador (generado internamente). Las reglas esqueleto no son tenidas en cuenta para el proceso de detección y disparo de reglas, éstas son utilizadas simplemente como plantillas para construir reglas completas.

Por defecto una instancia de una regla esqueleto permanece en la base de datos hasta que sea borrada por el usuario que la creó. Sin embargo, es posible establecer que el sistema la borre automáticamente luego de que la regla se dispare. Esto se indica al momento de crear la regla esqueleto con la cláusula *FIREDROP*. Esto automatiza el borrado de instancias que cumplen su vida útil en aplicaciones como *workflow*.

C) TIPOS DE EVENTOS SOPORTADOS

Los eventos primitivos que pueden ser definidos en *Arena* son:

- Explícito: Eventos que pueden ser definidos por el usuario. Estos eventos poseen un identificador y cero o más parámetros definidos por el usuario según sus requerimientos.

  Ejemplo:
  CREATE EXPLICIT EVENT ShutDown (maquina *VARCHAR*, nivel *NUMBER*)

- Enmascarado: Evento que estipula condiciones (mascara) sobre los parámetros de un evento explícito previamente creado. El evento sucede si el evento explícito sucede y sus parámetros cumplen con todas las condiciones estipuladas por la mascara.
  Ejemplo:
  CREATE MASKED EVENT RetirosAltos ON RetiroBancario {ValorRetiro > 500.000 }

- Enmascarado esqueleto: Son eventos enmascarados cuya mascara posee variables, es decir, especifica condiciones sobre parámetros pero con valores aun no conocidos. Estos valores pueden ser instanciados posteriormente.
  Ejemplo: El evento utiliza la variable ValorRetiro
  CREATE MASKED EVENT RetirosAltos ON RetiroBancario {ValorRetiro > $ValorRetiro }

- Temporal Absoluto: Eventos de tiempo que estipulan una fecha fija. El evento sucede cuando dicha fecha suceda.
  Ejemplo:
  CREATE TEMPORAL EVENT FechaDeCierre ON [10/07/98 10:00:00]

- Temporal Relativo: Evento de tiempo que estipula la ocurrencia de un lapso de tiempo con respecto a la ocurrencia de otro evento.

  Ejemplo:

  CREATE TEMPORAL EVENT TiempoDeReposo ON (MotorApagado) + 12 Minutes

D) OPERADORES DE EVENTOS

Los operadores de eventos permiten construir eventos compuestos. Un evento compuesto puede ser una expresión de eventos primitivos y/o de eventos compuestos. Los siguientes operadores y su semántica son soportados en *Arena*:

- Conjunción *(AND) (A AND B)* sucede cuando A y B suceden
- Disyunción *(OR) (A OR B)* sucede cuando A o B suceden
- Secuencia *(SEQ) (A SEQ B)* sucede cuando sucede A y luego sucede B
- Negación *(NOT) NOT(A, B, C)* sucede cuando B no sucede entre la ocurrencia de A y C.

La precedencia de los operadores es la siguiente: *NOT* > *AND* > *SEQ* > *OR*. El siguiente comando crea un evento compuesto:

CREATE COMPOSITE EVENT Alerta ON CorteFluidoElectrico SEQ CaidaMaquina

E) CONDICIÓN

La condición de las reglas en *Arena* puede tener una de las siguientes expresiones:

- Una expresión lógica sobre los parámetros de los eventos explícitos estipulados en el evento de la regla
- *TRUE*, es decir, la condición siempre evalúa a verdadero.

F) EXPRESIÓN DE PARÁMETROS

La condición puede construir una expresión con los operadores lógicos *AND, OR*, y *NOT* sobre los parámetros de los eventos explícitos estipulados en el evento de la regla.

Ejemplo:
ON shutdown IF shutdown.maquina = `neptuno' AND shutdown.mode = `normal' DO... // acción de la regla.

El siguiente comando crea una condición en el sistema de reglas:

CREATE CONDITION Condición1 AS shutdown.maquina = `neptuno'
AND shutdown.mode = `normal'

G) ACCIÓN

La acción de la regla debe hacer el llamado de al menos un procedimiento almacenado en un paquete del usuario dueño de la regla. Los argumentos del procedimiento llamado pueden ser valores constantes, parámetros de los eventos explícitos estipulados en el evento de la regla, o variables. Existe un procedimiento especial cuyo nombre es IR y sirve para instanciar reglas.

Los argumentos de este procedimiento son el identificador de la regla esqueleto a instanciar y una cadena de instanciación en la cual se instancian las variables que posea la regla ya sea en: Máscaras esqueleto o acciones esqueleto.

Ejemplo:
IR (`R8', 'nombre="Carlos", edad=25');

H) ACCIONES ESQUELETO

Es posible utilizar variables como argumentos en el llamado a un procedimiento. Cuando la acción de una regla referencia uno o más procedimientos y utiliza al menos una variable como argumento en alguno de los procedimientos (argumento esqueleto), se dice que la acción es una acción esqueleto. Este tipo de aciones permiten crear reglas más flexibles ya que pueden ser instanciadas en tiempo de ejecución.

Ejemplo:

ON compra *IF* compra.valor > 1.000.000 DO mipaquete.descuentos(compra.numero_factura, $porcentaje_descuento);

El siguiente comando crea una nueva acción esqueleto:

CREATE ACCION Accion1 AS mipaquete.abonar($cuenta, $valor)

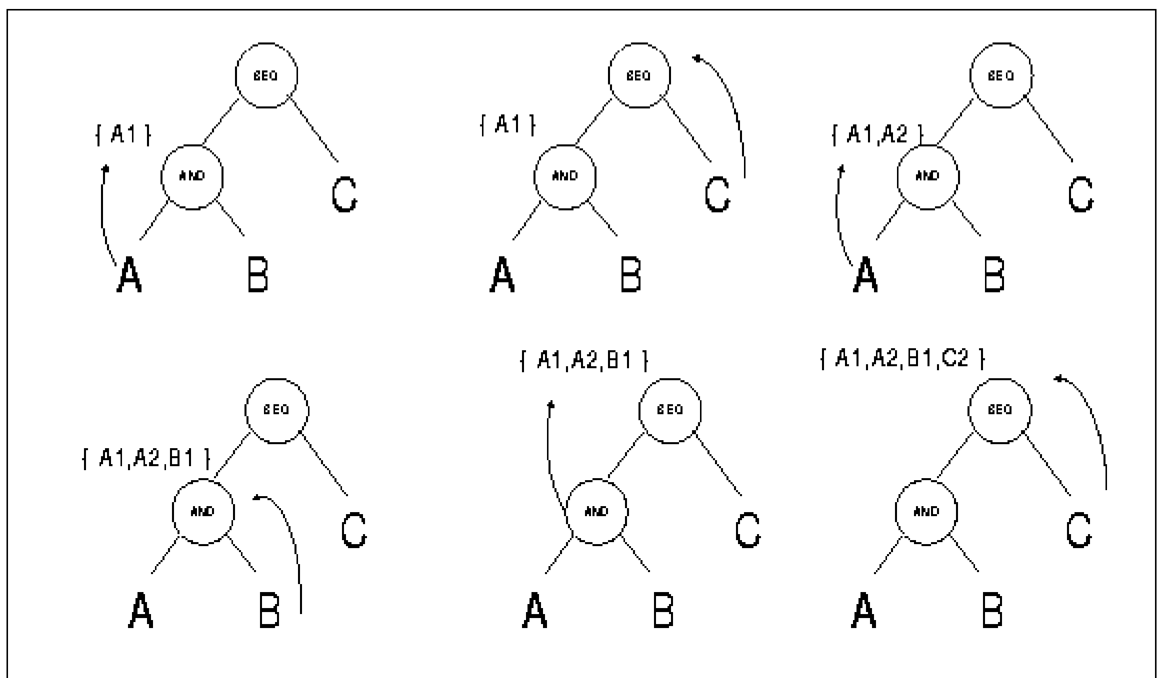
I) DETECCIÓN DE EVENTOS

Roke utiliza un algoritmo basado en árboles como técnica para la detección de eventos compuestos. Dado un evento E = A *AND* B *SEQ* C, donde A, B y C pueden ser eventos primitivos o compuestos, su representación en forma de árbol y su detección se deriva de la siguiente manera.

Los nodos terminales del árbol representan eventos primitivos o compuestos, y los eventos no terminales representan operadores de eventos. La ubicación de cada operador en el árbol depende de su precedencia. Cada nodo no terminal es proveído de una memoria que le permite recordar todas las instancias de eventos que hicieron detectar a sus nodos hijos (inmediatos).

Cada vez que se cumple la semántica de un operador en un nodo no terminal, el contenido de su memoria es transferido a su nodo padre, quedando la memoria del nodo detectado nuevamente vacía. El evento compuesto es detectado cuando se cumpla la semántica del operador del nodo raíz en el árbol. Dada como ejemplo una historia de eventos H = {A1, C1, A2, B1, C2}, el proceso de detección del evento compuesto E puede ser apreciado en la Figura 4.

Figura 4. Detección de eventos



Nótese como la instancia C1 es rechazada por el nodo del operador *SEQ* debido a que su semántica solo acepta la ocurrencia de su hijo izquierdo en primer lugar y luego la de su hijo derecho.

3.2.1.4 Coordinación de procesos de negocio

Una de las formas de llevar a cabo los procesos de una compañía es utilizando herramientas *workflow*. Un sistema *workflow* es un producto de software capaz de coordinar la ejecución de los procesos del negocio o *workflows*.

Un *workflow* es un conjunto de actividades o tareas que poseen un orden de ejecución y que son desarrolladas por múltiples agentes humanos y/o agentes autómatas.

Uno de las técnicas propuestas para la coordinación de procesos de *workflow* es el uso de reglas activas. No obstante, aquí se propone un enfoque según el cual la herramienta de *workflow* se encuentra ligeramente acoplada con una subyacente. En la se define un conjunto de reglas esqueleto para cada *workflow*. Cada vez que un nuevo caso o instancia de *workflow* inicia, las reglas esqueleto para dicho *workflow* se van instanciando para coordinar la ejecución de ese caso particular.

En un momento dado muchos casos de un *workflow* pueden estar en ejecución. Adicionalmente se propone que las reglas para un caso no sean instanciadas en su totalidad desde el principio del caso, sino que dicha instanciación se haga progresivamente a medida que el caso se desarrolla.

Una actividad o tarea es modelada como un evento explícito cuyos parámetros corresponden a información propia de la tarea tales como el identificador del *workflow* al que pertenece, el agente que la realizó, la instancia del *workflow* (caso *workflow*), documentos producidos, datos que permiten tomar decisiones sobre cuál debe ser la siguiente tarea a realizar y la fecha de terminación de la tarea.

Un prototipo de herramienta *workflow* denominada *RuleFlow* ha sido desarrollado para implementar la propuesta anterior. Se supone que existe una herramienta que es capaz de generar las reglas esqueleto de un *workflow* con base en su especificación gráfica. Se propone adoptar los estándares de la *Work Flow Management Coalition* en cuanto a las estructuras de control que se deben utilizar para especificar el encadenamiento de actividades en un *workflow.*

A) ARQUITECTURA *RULEFLOW*

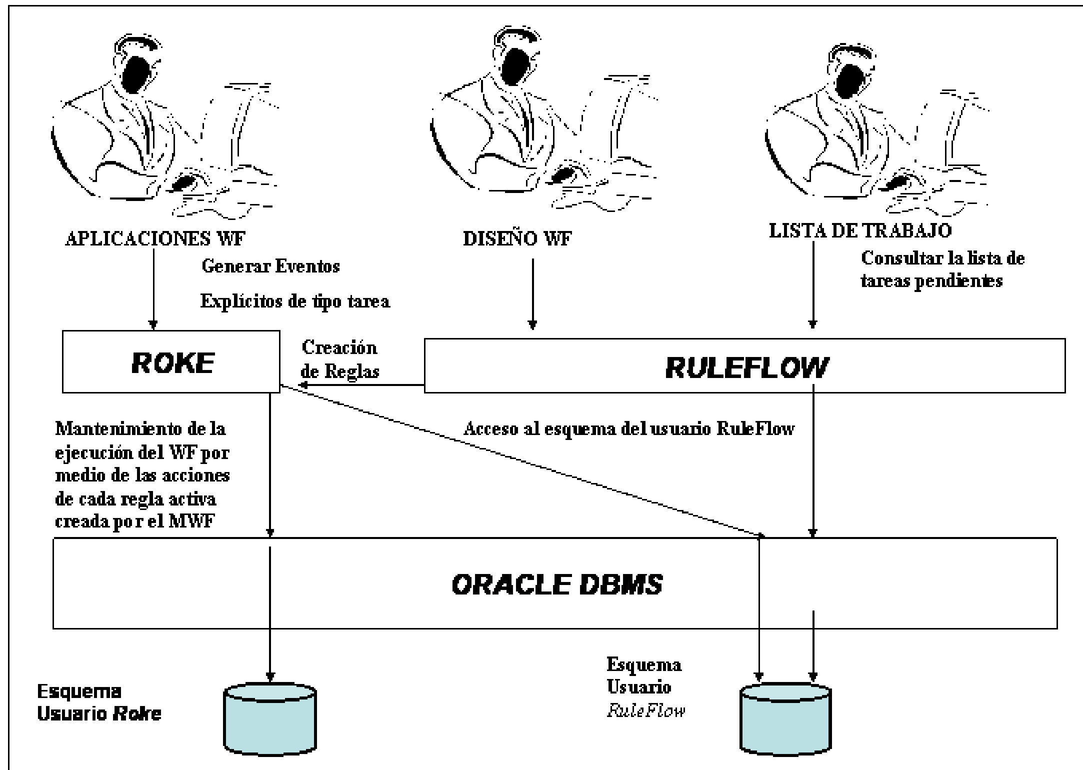
El diseño de los procesos de *workflow* es realizado mediante un módulo de *RuleFlow* capaz de convertir un diseño gráfico de un proceso *workflow* en un conjunto de reglas activas que coordinan o conducen dicho proceso. La información gráfica y definición de cada proceso, cuántos casos se encuentran en ejecución, listas de trabajo y mensajes a los agentes son almacenados en el esquema de un usuario *Orácle* denominado *Ruleflow*.

Las reglas generadas son introducidas y compiladas en el ambiente *Roke*. Estas reglas quedan almacenadas dentro del esquema del usuario *Roke*. Las aplicaciones *workflow* utilizan el procedimiento *RaiseEvent* del ambiente *Roke* para generar eventos explícitos que indiquen la terminación de cada tarea desarrollada por los agentes del *workflow*.

Las reglas activas que coordinan los procesos del *workflow* reaccionan ante los eventos explícitos que representan la finalización de tareas. La acción de las reglas va actualizando el estado de la ejecución de todos los casos del proceso *workflow* directamente en las estructuras de datos establecidas en el esquema del usuario *RuleFlow*.

RuleFlow provee una herramienta para que los agentes consulten los mensajes y la lista de tareas (*worklist*) que tienen pendientes por realizar. Esta herramienta accede a las estructuras de datos que almacenan dicha información dentro del esquema del usuario *RuleFlow*. La arquitectura de *RuleFlow* es mostrada en la figura 5.

Figura 5. Arquitectura *Ruleflow*



B) Coordinación de procesos *workflow*

La coordinación de cada instancia de *workflow* se efectúa de la siguiente manera. Un agente reporta la creación de un nuevo caso generando un evento explícito especial denominado NuevoCaso. Todo *workflow* posee una regla completa que reacciona frente a este evento explícito (R_NuevoCaso).

Ejemplo:

CREATE RULE R25_NuevoCaso ON NuevoCaso IF NuevoCaso.wf = 25 DO ruleflow.na (25,NuevoCaso.caso, `RECEPCION'); ruleflow.nc (25, NuevoCaso.caso, NuevoCaso.toc, NuevoCaso.agente);
IR(R25_Recepcion, `caso=NuevoCaso.caso');

La condición de la regla evalúa si el nuevo caso es del *workflow* al que dicha regla pertenece, y si es así, ejecuta su acción la cual notifica en las listas de trabajo (mediante el procedimiento *Ruleflow.na*) a un conjunto de agentes que la primer tarea de un nuevo caso está pendiente de realizar.

La regla también registra la información del nuevo caso utilizando el procedimiento *ruleflow.nc*. Por último crea una instancia de la regla R25_Recepcion, la cual se encargará de reaccionar frente a la realización de la primera tarea ya notificada.

Suponiendo que el nuevo caso generado tiene identificador 14, la regla R25_Recepcion es instanciada para reaccionar frente a la ocurrencia de la terminación de la tarea Recepción para el caso 14. Los agentes deben negociar internamente el encargado de realizar dicha tarea. Cuando el agente elegido realiza la tarea, la herramienta o aplicación con la cual desarrolla la tarea, genera un evento explícito en el ambiente *Roke.*

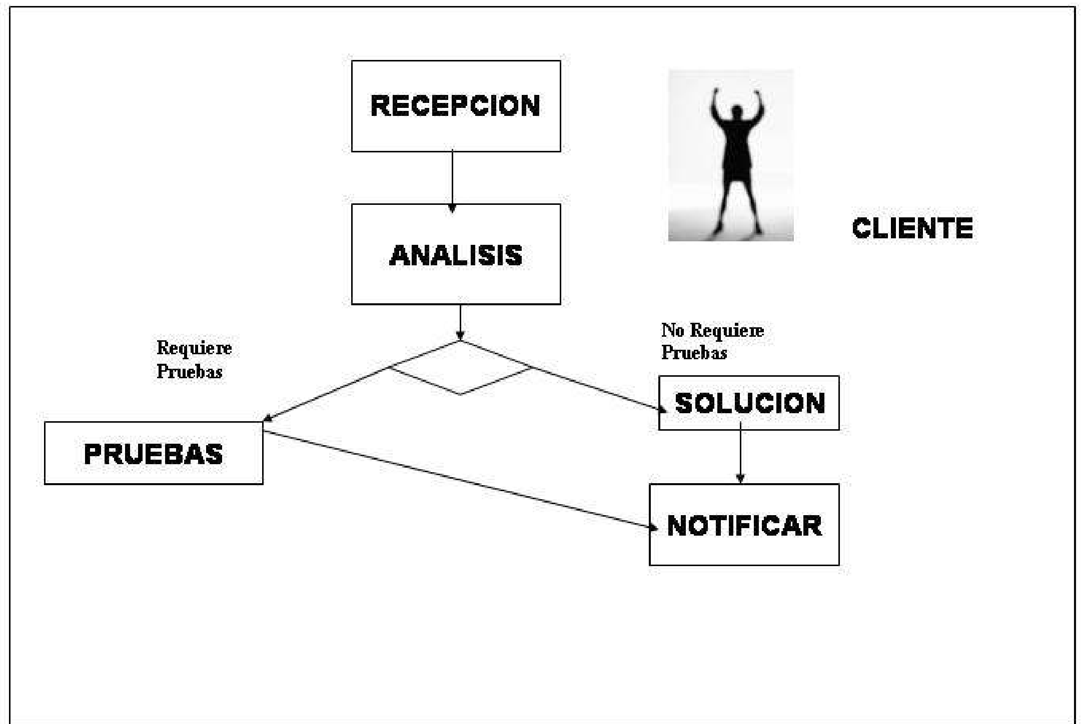
Indicando que la tarea finalizó. La regla R25_Recepcion reacciona ante el evento explícito correspondiente, utiliza su condición para evaluar los parámetros del evento explícito y con su acción estipula cuál es la siguiente tarea a realizar. Nuevamente, la acción de la regla actualiza el esquema del usuario *RuleFlow* sobre la ocurrencia de una nueva tarea, notifica a los agentes sobre la próxima tarea a realizar e instancia la regla o reglas necesarias para coordinar las tareas que deben ser realizadas a continuación.

Las reglas que reaccionan frente a tareas donde termina el *workflow* no crean instancias de reglas, simplemente registran la terminación del caso, agente que lo terminó y fecha de terminación. Los procedimientos que provee *RuleFlow* para insertar la información de un nuevo caso, notificar agentes, borrar tareas de la lista de trabajo y reportar un fin de caso son respectivamente: *Ruleflow.nc, Ruleflow.na, Ruleflow.ft, y Ruleflow.fc.*

3.2.1.5 Un caso práctico: Proceso de atención de reclamos

Un proceso de atención al cliente para la recepción de reclamos es especificado en la figura 6. El identificador asignado a dicho *workflow* es el número 25. La primera actividad a realizar es recibir el reclamo del cliente y luego efectuar un análisis preliminar del reclamo. En ese análisis se decide si el reclamo requiere pruebas adicionales o no. Si se requieren pruebas, una tarea de solicitud y evaluación de pruebas es realizada. Luego de tener las pruebas o en el caso de no requerir pruebas, el siguiente paso es solucionar el reclamo. Cuando el reclamo es solucionado ya sea a favor o en contra el cliente, la decisión es notificada al cliente.

Figura 6. Proceso de atención de reclamos



CREATE RULE R25_NuevoCaso ON NuevoCaso IF NuevoCaso.wf = 25
DO ruleflow.na (25, NuevoCaso.caso, `RECEPCION');
ruleflow.nc (25, NuevoCaso.caso, NuevoCaso.agente, NuevoCaso.toc);
IR (`R25_Recepcion', `caso=NuevoCaso.caso');

CREATE SKELETON RULE R25_Recepcion
ON Recepcion {wf=25, caso=$caso} IF TRUE
DO ruleflow.ft (25, Recepcion.caso, `RECEPCION',
Recepcion.agente, Recepcion.toc);
ruleflow.na (25, Recepcion.caso, `ANALISIS');
IR (`R25_Analisis', `caso=Recepcion.caso');

CREATE SKELETON RULE R25-1_Analisis ON Analisis {wf=25, caso=$caso}
IF Analisis.pruebas = `SI' DO ruleflow.ft (25, Analisis.caso, `ANALISIS',
Analisis.agente, Analisis.toc);
ruleflow.na (25, Analisis.caso, `PRUEBAS');
IR (`R25_Pruebas', `caso=Analisis.caso');

CREATE SKELETON RULE R25-2_Analisis ON Analisis {wf=25, caso=$caso}
IF Analisis.pruebas = `NO' DO ruleflow.ft (25, Analisis.caso, `ANALISIS',
Analisis.agente, Analisis.toc);
ruleflow.na (25, Analisis.caso, `SOLUCION');
IR (R25_Solucion, `caso=Analisis.caso');

CREATE SKELETON RULE R25_Pruebas ON Pruebas {wf=25, caso=$caso}
IF TRUE DO ruleflow.ft (25, Pruebas.caso, `PRUEBAS', Pruebas.agente,
Pruebas.toc);
ruleflow.na (25, Pruebas.caso, `SOLUCION');
IR (R25_Solucion, `caso=Pruebas.caso');

CREATE SKELETON RULE R_Solucion ON Solucion {wf=25, caso=$caso}
IF TRUE DO ruleflow.ft (25, Solucion.caso, `SOLUCION',
Solucion.agente, Solucion.toc);
ruleflow.na (25, Solucion.caso, `NOTIFICAR');
IR (R25_Notificar, `caso=Solucion.caso');

CREATE SKELETON RULE R_Notificar ON Notificar {wf=25, caso=$caso}
IF TRUE DO ruleflow.ft (25, Notificar.caso, `NOTIFICAR',
Notificar.agente, Notificar.toc);
ruleflow.fc (25, Notificar.caso, Notificar.agente, Notificar.toc);

3.2.2 *Db2*

La mayoría de los sistemas de base de datos relacionales proporcionan soporte para los disparadores. IBM agrego soporte de disparadores a *DB2* para *OS/390* en la versión 6 (anunciada en mayo de 1998, para estar generalmente disponible en junio de 1999). Por facilidad de especificación se utilizara indistintamente *trigger* y disparador.

3.2.2.1 Sintaxis de *Db2 Create Trigger*

CREATE TRIGGER nombre
{ BEFORE | AFTER} evento *OF [*campos*]*
ON Tabla
FOR EACH ROW MODE DB2SQL [SENTECIAS]

3.2.2.2 Semántica de disparadores

Los disparadores se pueden poner en ejecución para muchas aplicaciones prácticas. Es absolutamente a menudo imposible cifrar reglas de negocio en la base de datos usando solamente *LDD*. Por ejemplo, *DB2* no utiliza restricciones complejas (solamente restricciones basadas en un solo valor) o los varios tipos de restricciones de referencia (tales como *DELETE* que se procesa en cascada o *UPDATE*).

Usando disparadores, un ambiente más flexible se establece para poner reglas de negocio y restricciones en ejecución en el DBMS. Esto es importante, porque al tener las reglas de negocio en la base de datos se asegura que cada uno utiliza la misma lógica para lograr el mismo proceso. Los disparadores se pueden codificar para tener acceso y/o para modificar a otras tablas, imprimir mensajes, y especifican restricciones complejas. Dos opciones existen cuando se ejecuta un disparador en *Db2*:

"Antes", se ejecuta antes de que ocurra una actividad, en ingles *berofe.*
"Después", se ejecuta después de que ocurra una actividad, en ingles *"after".*

En *Db2* Versión 6, con la opción " *before*" los disparadores son restringidos porque no pueden realizar actualizaciones. Conociendo la función de los disparadores en una base de datos estos son imprescindibles.

Una característica interesante de los disparadores de *Db2* V6 es el orden en el cual se disparan. ¿Si múltiples disparadores son codificados para una misma tabla, qué disparador se enciende primero?. Puede diferenciarse en cómo los disparadores son codificados, son probados, y son mantenidos. La regla para el orden de la ejecución es básicamente simple de entender, pero puede ser difícil de mantener. Para los disparadores del mismo tipo, se ejecutan en el orden en el cual fueron creados. Por ejemplo, si dos disparadores "*delete*" se codifican para la misma tabla, el que fue creado físicamente primero, se ejecuta primero.

3.2.2.3 Usando disparadores para implementar integridad referencial

Uno de los usos primarios de los disparadores (*triggers*) es dar soporte a la integridad referencial (IR). *Db2* da soporte para una robusta norma de declaración de IR. Los DBMS completos dan soporte para todas las posibles restricciones referenciales. En la tabla III esta la lista de las posibilidades que presenta *Db2*.

Tabla III. Reglas de integridad referencial

Regla de integridad	Descripción
DELETE RESTRICT	Si existe alguna fila en la tabla dependiente, la llave primaria de la fila en la tabla padre no puede ser borrada.
DELETE CASCADE	Si existe alguna fila en la tabla dependiente, la llave primaria de la fila en la tabla padre puede ser borrada, y todas las filas dependientes también son borradas.
DELETE NEUTRALIZE	Si existe alguna fila en la tabla dependiente, la llave primaria de la fila en la tabla padre puede ser borrada, y la Llave extranjera para todas las filas dependientes se coloca el valor de *NULL*.
UPDATE RESTRICT	Si existe alguna fila en la tabla dependiente, la columna de la llave primaria de la fila en la tabla padre no puede ser actualizada
UPDATE CASCADE	Si existe alguna fila en la tabla dependiente, la columna de la llave primaria de la fila en la tabla padre no puede ser actualizada
UPDATE NEUTRALIZE	Si existe alguna fila en la tabla dependiente, las columnas que forman la llave primaria de la fila en la tabla padre puede ser actualizada, y todas las filas dependientes también son actualizadas.
INSERT RESTRICT	Un valor de una llave extranjera no puede ser insertado dentro de una tabla dependiente a menos que el valor de la llave primaria ya exista en la tabla padre.
FK UPDATE	Un valor de una llave extranjera no puede ser actualizado una tabla dependiente a menos que el valor de la llave primaria ya exista en la tabla padre.
PENDANT DELETE	Cuando el ultimo valor de la llave extranjera de la tabla dependiente es borrado la llave primaria de la fila en la tabla padre también es borrado.

El orden para usar disparadores para dar soporte a reglas IR es a veces necesario, para conocer el impacto de la acción que se ejecuta al dispararse el *trigger*.

3.2.2.4 Ejemplos de ejecución de disparadores

Este es un disparador de actualización, codificado para la tabla EMP. Este *trigger* implementa un simple chequeo antes de realizar la actualización, este chequeo consiste en verificar que el nuevo salario no exceda el 50%, cuando el nuevo salario excede el 50 % del salario normal es llamada una rutina de error.

```
CREATE TRIGGER SALARY_UPDATE
BEFORE UPDATE OF SALARY  ON EMP
FOR EACH ROW MODE DB2SQL
WHEN (NEW.SALARY > (OLD.SALARY * 1.5))
     BEGIN ATOMIC
      SIGNAL SQLSTATE  '75001' ('Raise exceeds 50%');
 END;
```

El disparador se ejecuta una vez por cada fila. Si múltiples filas son modificadas por una simple actualización el disparador se ejecuta en múltiples tiempos, modificando una fila a la vez.

3.2.3 *Sql3*

Las propuestas de Codd en 1970 relativas al Modelo Relacional (MR) incentivaron a las universidades y centros de investigación a diseñar lenguajes que soporten este modelo. El lenguaje relacional *SQL* fue desarrollado originalmente por IBM en un sistema prototipo llamado “Sistema R”, con el nombre de *SEQUEL*, en los años 1974-1975. Posteriormente por razones legales, pasó a llamarse *SQL* (por sus siglas en ingles: *Structured Query Language,* Lenguaje de Consulta Estructurado). En 1979 surgieron los primeros productos comerciales que lo implementaron (*SQL Orácle*). Después de casi 13 años, en 1992, el lenguaje *SQL* fue finalmente estandarizado por los grupos ANSI e ISO, llamándose *SQL*-92 o *SQL2*.

La función del lenguaje *SQL* es la de soportar la definición, manipulación y control de los datos en una base de datos relacional. Desde el punto de vista de este lenguaje una base de datos relacional es un conjunto de tablas, donde cada tabla es un conjunto de filas y columnas.

SQL es mucho más que un lenguaje de consulta de base de datos, ya que permite:

- Definir la estructura de los datos
- Recuperar y manipular datos
- Administrar y controlar el acceso a los datos
- Compartir datos de forma concurrente
- Asegurar la integridad

El término integridad en las bases de datos se refiere a asegurar que los datos sean válidos. La especificación de reglas de integridad (llamadas también reglas del negocio) es relativamente compleja, y no era posible definirlas como elementos de la estructura de la base de datos. Era deseable que los DBMS almacenaran las reglas de integridad en el diccionario de datos, de forma que el subsistema de integridad esté constantemente monitoreando las transacciones que realizan los usuarios, asegurando que las reglas se cumplan. Esto es posible actualmente por medio de las Bases de Datos Activas.

3.2.3.1 Características activas en *Sql3*

Muchos fabricantes de Sistemas de administración de base de datos relacionales incorporan algunas funcionalidades propias de Bases de Datos Activas en sus productos, consistentes en el uso de disparadores. En la versión de *SQL (SQL3*) se incluyen los estándares de definición y utilización de disparadores.

3.2.3.1.1 Disparadores en Sql3

Existe una propuesta para la definición de disparadores para el estándar de *SQL*. A continuación se presenta la sintaxis.

CREATE TRIGGER nombre_disparador
{BEFORE | AFTER} {INSERT | DELETE | UPDATE | UPDATE OF col1, col2...}
ON relación [referencia][condición]
Acciones [granularidad]

El disparador se activa antes o después de una modificación sobre la relación (inserción, borrado, actualización) o actualización de unas columnas determinadas. Las opciones son las siguientes:

Referencia. Sólo en actualizaciones, es la manera de indicar cuál es el valor antiguo y el nuevo de las tuplas tras la actualización.

REFERENCING OLD [AS] n_antiguo [NEW AS n_nuevo] o a la inversa.

Condición: Indica una condición necesaria para la activación del disparador. Si se omite se activa directamente.

WHEN (expresión)

Acciones: Indica las acciones (separadas por comas) del disparador.

Granularidad*:* Indica si la acción se ejecuta una vez por cada evento o bien una vez por cada tupla a la que se refiere el evento.

3.2.4 *Postgres*

3.2.4.1 ¿Qué es *Postgres*?

Los sistemas de mantenimiento de bases de datos relacionales tradicionales (DBMS) soportan un modelo de datos que consisten en una colección de relaciones con nombre, que contienen atributos de un tipo específico. En los sistemas comerciales actuales, los tipos posibles incluyen numéricos de punto flotante, enteros, cadenas de caracteres, cantidades monetarias y fechas. Está generalmente reconocido que este modelo será inadecuado para las aplicaciones futuras de procesado de datos. El modelo relacional sustituyó modelos previos en parte por su "simplicidad". Sin embargo, como se ha mencionado, esta simplicidad también hace muy difícil la implementación de ciertas aplicaciones. *Postgres* ofrece una característica adicional al incorporar los siguientes cuatro conceptos básicos en una vía en la que los usuarios pueden extender fácilmente el sistema.

- Clases
- herencia
- tipos
- funciones

Otras características que aporta *Postgres* son:

- Restricciones (*Constraints*)
- Disparadores (*triggers*)
- Reglas
- Integridad transaccional

Estas características colocan a *Postgres* en la categoría de las bases de datos identificadas como objeto-relacionales. Nótese que éstas son diferentes de las referidas como orientadas a objetos, que en general no son bien aprovechables para soportar lenguajes de bases de datos relacionales tradicionales. De hecho, algunas bases de datos comerciales han incorporado recientemente características en las que *Postgres* fue pionera.

3.2.4.2 El Sistema de reglas de *Postgres*

Los sistemas de reglas de producción son conceptualmente simples, pero hay muchos puntos sutiles implicados en el uso actual de ellos. Las reglas de Base de Datos Activa se implementan en *Postgres* como funciones y disparadores. El sistema de reglas de reescritura de consultas (el "sistema de reglas") es totalmente diferente a los procedimientos almacenados y a los disparadores. Este sistema modifica las consultas para tomar en consideración las reglas y entonces pasa la consulta modificada al optimizador para su ejecución. Es muy poderoso, y puede utilizarse de muchas formas, tales como procedimientos, vistas y versiones del lenguaje de consulta.

3.2.4.2.1 ¿Qué es un árbol de consulta?

Es una representación interna de una instrucción *Sql* donde se almacenan de modo separado las partes menores que la componen. Estos árboles de consultas son visibles cuando se arranca el motor de Postgres con nivel de *debug 4* y se teclea *queries* en la interface de usuario interactivo. Las acciones de las reglas almacenadas en el catalogo del sistema *pg_rewrite* están almacenadas también como árboles de consultas.

Las partes de un árbol de consulta son:

- El tipo de comando *(commandtype).* Este es un valor sencillo que nos dice el comando que produjo el árbol de traducción (*select, insert, update, delete*).
- La tabla de rango *(rangetable).* La tabla de rango es una lista de las relaciones que se utilizan en la consulta. En una instrucción *Select*, son las relaciones dadas tras la palabra clave *from*. Toda entrada en la tabla del rango identifica una tabla o vista, y dice el nombre por el que se identifica en las otras partes de la consulta. En un árbol de consulta, las entradas de la tabla de rango se indican por un índice en lugar de por su nombre como estarían en una instrucción *Sql*. Esto puede ocurrir cuando se han mezclado las tablas de rangos de reglas.
- La relación-resultado *(resultrelation).* Un índice a la tabla de rango que identifica la relación donde irán los resultados de la consulta. Las consultas *Select* normalmente no tienen una relación resultado. El caso especial de una *"select into"* es principalmente idéntica a una secuencia *Create Table, Insert...Select*. En las consultas *insert, update y delete*, la relación resultado es la tabla (o vista) donde tendrán efecto los cambios.
- La lista objetivo *(targetlist)* La lista objetivo es una lista de expresiones que definen el resultado de la consulta. En el caso de una instrucción *select*, las expresiones son las que construyen la salida final de la consulta. Son las expresiones entre las palabras clave *select y from*.

Las consultas "*delete*" no necesitan una lista objetivo porque no producen ningún resultado. De hecho, el optimizador añadirá una entrada especial para una lista objetivo vacía. Para el sistema de reglas, la lista objetivo está vacía.

En Consultas "*insert*" la lista objetivo describe las nuevas filas que irán a la relación resultado. Las columnas que no aparecen en la relación resultado serán añadidas por el optimizador con una expresión constante *NULL*. Son las expresiones de la cláusula "*values*" y las de la cláusula *Select* en una *Insert … Select*.

En consultas "*update*", describe las nuevas filas que reemplazarán a otras antiguas. Ahora el optimizador añadirá las columnas que no aparecen, insertando expresiones que recuperan los valores de las filas antiguas en las nuevas, y añadirá una entrada especial como lo hace *delete*. Es la parte de la consulta que recoge las expresiones del atributo set atributo = expresión.

Cada entrada de la lista objetivo contiene una expresión que puede ser un valor constante, una variable apuntando a un atributo de una de las relaciones en la tabla de rango, un parámetro o un árbol de expresiones de llamadas a funciones, constantes, variables, operadores, etc.

- La cualificación. La cualificación de las consultas es una expresión muy similar a otra de las contenidas en las entradas de la lista objetivo. El valor resultado de esta expresión es un booleano que dice si la operación (*insert, update, delete* o *select*) para las filas del resultado final deberá ser ejecutada o no. Es la cláusula *where* de una instrucción *SQL*.

3.2.4.2.2 Las vistas y el sistema de reglas

Las vistas en *Postgres* se implementan utilizando el sistema de reglas. De hecho, no hay diferencia entre:

*CREATE VIEW mi_vista AS SELECT * FROM mi_tabla;* y la secuencia:
CREATE TABLE mi_vista (la misma lista de atributos de mi_tabla);
CREATE RULE "_RETmi_vista" AS ON SELECT TO mi_vista DO INSTEAD
*SELECT * FROM mi_tabla;*

Esto es exactamente lo que hace internamente el comando *Create View*. Esto tiene algunos efectos colaterales. Uno de ellos es que la información sobre una vista en el sistema de *Postgres* es exactamente el mismo que para una tabla. De este modo, para los traductores de consultas, no hay diferencia entre una tabla y una vista.

3.2.4.2.2.1 Cómo trabajan las reglas de *select*

Las reglas *ON SELECT* se aplican a todas las consultas, incluso si el comando dado es *insert, update o delete* y tienen diferentes semánticas de las otras en las que se modifica el árbol de traducción en lugar de crear uno nuevo.

El ejemplo son dos vistas unidas que hacen algunos cálculos y algunas otras vistas utilizadas para ello. Una de estas dos primeras vistas se personaliza añadiendo reglas para operaciones de *insert, update y delete* de modo que el resultado final será una vista que se comporta como una tabla real con algunas otras funcionalidades. Las tablas y vistas que se utilizaran en la descripción del sistema de reglas son:

```
CREATE TABLE shoe_data (      -- datos de zapatos
   shoename   char(10),      -- clave primaria (primary key)
   sh_avail   integer,       -- número de pares utilizables
   slcolor    char(10),      -- color de cordón preferido
   slminlen   float,         -- longitud mínima de cordón
   slmaxlen   float,         -- longitud máxima del cordón
   slunit     char(8)        -- unidad de longitud
);

CREATE TABLE shoelace_data (  -- datos de cordones de zapatos
   sl_name    char(10),      -- clave primaria (primary key)
   sl_avail   integer,       -- número de pares utilizables
   sl_color   char(10),      -- color del cordón
   sl_len     float,         -- longitud del cordón
   sl_unit    char(8)        -- unidad de longitud
);

CREATE TABLE unit (           -- unidades de longitud
   un_name    char(8),       -- clave primaria (primary key)
   un_fact    float          -- factor de transformación a cm
);

CREATE VIEW shoe AS
 SELECT sh.shoename,  sh.sh_avail,  sh.slcolor,  sh.slminlen,
 sh.slminlen * un.un_fact AS slminlen_cm,
 sh.slmaxlen,  sh.slmaxlen * un.un_fact AS slmaxlen_cm, sh.slunit
 FROM shoe_data sh, unit un
 WHERE sh.slunit = un.un_name;
```

```
CREATE VIEW shoelace AS
SELECT s.sl_name, s.sl_avail, s.sl_color, s.sl_len,  s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
SELECT rsh.shoename,  rsh.sh_avail,rsl.sl_name,  rsl.sl_avail,  min(rsh.sh_avail, rsl.sl_avail)
AS total_avail
FROM shoe rsh, shoelace rsl WHERE rsl.sl_color = rsh.slcolor
AND rsl.sl_len_cm >= rsh.slminlen_cm  AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

El comando *Create View* para la vista *shoelace* creará una relación shoelace y una entrada en *pg_rewrite* que dice que hay una regla de reescritura que debe ser aplicada siempre que la relación shoelace sea referida en la tabla de rango de una consulta.

```
al_bundy=> SELECT * FROM shoelace;
```

La consulta anterior es sencilla, y se usara para explicar las reglas de las vistas. *"SELECT * FROM shoelace*" fue interpretado por el traductor y produjo un árbol de traducción y se le asigna al sistema de reglas.

```
SELECT shoelace.sl_name, shoelace.sl_avail,
        shoelace.sl_color, shoelace.sl_len,
        shoelace.sl_unit, shoelace.sl_len_cm
   FROM shoelace shoelace;
```

El sistema de reglas revisa la tabla de rango, y comprueba si hay reglas en *pg_rewrite* para alguna relación. Cuando se procesa las entradas en la tabla de rango para *shoelace* encuentra la regla '_RETshoelace' con el árbol de traducción

```
SELECT s.sl_name, s.sl_avail, s.sl_color, s.sl_len, s.sl_unit,
 float8mul(s.sl_len, u.un_fact) AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*, shoelace_data s, unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

Nótese que el traductor cambió el cálculo y la cualificación en llamadas a las funciones apropiadas. Pero de hecho esto no cambia nada. El primer paso en la reescritura es mezclar las dos tablas de rango. El árbol de traducción entonces lee:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*,  shoelace_data s,  unit u;
```

En el segundo paso, añade la cualificación de la acción de las reglas al árbol de traducción, el resultado es:

```
SELECT shoelace.sl_name, shoelace.sl_avail, shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace, shoelace *OLD*, shoelace *NEW*, shoelace_data s, unit u
WHERE bpchareq(s.sl_unit, u.un_name);
```

En el paso tres, reemplaza todas las variables en el árbol de traducción, que se refieren a entradas de la tabla de rango por las correspondientes expresiones de la lista objetivo correspondiente a la acción de las reglas.

El resultado es la consulta final:

```
SELECT s.sl_name, s.sl_avail,  s.sl_color, s.sl_len, s.sl_unit, float8mul(s.sl_len, u.un_fact) AS
sl_len_cm
FROM shoelace shoelace, shoelace *OLD*,  shoelace *NEW*, shoelace_data s,  unit u
WHERE bpchareq(s.sl_unit, u.un_name);
```

Para realizar esta salida en una instrucción SQL real, un usuario debería teclear:

```
SELECT s.sl_name, s.sl_avail, s.sl_color, s.sl_len, s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;
```

Esta ha sido la primera regla aplicada. Mientras se iba haciendo esto, la tabla de rango iba creciendo. De modo que el sistema de reglas continúa comprobando las entradas de la tabla de rango. Lo siguiente es el número 2 (*shoelace *OLD*).* La Relación *shoelace* tiene una regla, pero su entrada en la tabla de rangos no está referenciada en ninguna de las variables del árbol de traducción, de modo que se ignora.

Puesto que todas las entradas restantes en la tabla de rango, o bien no tienen reglas en *pg_rewrite* o bien no han sido referenciadas, se alcanza el final de la tabla de rango.

La reescritura está completa y el resultado final dado se pasa al optimizador. El optimizador ignora las entradas extra en la tabla de rango que no están referenciadas por variables en el árbol de traducción, y el plan producido por el planificador/optimizador debería ser exactamente el mismo que si se hubiese tecleado la *SELECT* anterior en lugar de la selección de la vista.

3.2.4.2.2.2 El poder de las vistas en *Postgres*

Todo lo anterior demuestra como el sistema de reglas incorpora las definiciones de las vistas en el árbol de traducción original. Los beneficios de implementar las vistas con el sistema de reglas están en que el optimizador tiene toda la información sobre qué tablas tienen que ser revisadas, más las relaciones entre estas tablas, más las cualificaciones restrictivas a partir de la definición de las vistas, más las cualificaciones de la consulta original, todo en un único árbol de traducción. Y esta es también la situación cuando la consulta original es ya una unión entre vistas. El optimizador debe decidir cuál es la mejor ruta para ejecutar la consulta. Cuanta más información tenga el optimizador, mejor será la decisión y la forma en que se implementa el sistema de reglas en *Postgres* asegura que toda la información sobre la consulta está utilizable.

3.2.4.2.3 Reglas y permisos

Debido a la reescritura de las consultas por el sistema de reglas de *Postgres*, se han accedido a otras tablas/vistas diferentes a la consulta original. Utilizando las reglas de actualización (*update*), se puede permitir acceso sobre la escritura a tablas.

En las reglas de reescritura no se puede diferenciar un propietario. El propietario de una relación (tabla o vista) es automáticamente el propietario de las reglas de reescritura definidas para ella. El sistema de reglas de *Postgres* cambia el comportamiento del sistema de control de acceso de defecto. Las relaciones que se utilizan debido a las reglas son comprobadas durante la reescritura contra los permisos del propietario de la relación, contra la que la regla se ha definido. Esto hace que el usuario no necesite sólo permisos para las tablas/vistas a las que él hace referencia en sus consultas. Por ejemplo: Un usuario tiene una lista de números de teléfono en la que algunos son privados y otros son de interés para la secretaria en la oficina. Se puede construir lo siguiente:

CREATE TABLE agenda (persona text, telefono text, privado bool);
CREATE VIEW numero_telefono AS SELECT persona, teléfono FROM agenda
WHERE NOT privado;
GRANT SELECT ON numero_Telefono TO secretaria;

Nadie excepto el usuario, y el superusuario de la base de datos, pueden acceder a la tabla agenda. Pero debido a la instrucción *grant*, la secretaria puede hacer una consulta "*selec*t" a través de la vista numero_Telefono. El sistema de reglas reescribirá la consulta "*Select*" de numero_Telefono en una "*Select*" de *agenda* y añade la cualificación de que sólo se buscan las entradas cuyo campo "privado" sea falso. Una vez que el usuario sea el propietario de numero_Telefono, la lectura accede a agenda, se comprueba contra sus permisos, y la consulta se considera autorizada. La comprobación para acceder a numero_Telefono se realiza entonces, de modo que nadie más que la secretaria pueda utilizarlo.

Los permisos son comprobados regla a regla. De modo que la secretaria es ahora la única que puede ver los números de teléfono públicos. Pero la secretaria puede crear otra vista y autorizar el acceso al público. Entonces, cualquiera puede ver los datos de numero_Telefono a través de la vista de la secretaria.

Lo que la secretaria no puede hacer es crear una vista que acceda directamente a agenda (realmente sí puede, pero no trabajará, puesto que cada acceso abortará la transacción durante la comprobación de los permisos), y tan pronto como el usuario tenga noticia de que la secretaria ha abierto su vista a numero_Telefono, el usuario puede denegar el acceso por medio de *Revoke*, inmediatamente después, cualquier acceso a la vista de las secretarias fallará.

Se podría pensar que este chequeo regla a regla es un agujero de seguridad, pero de hecho no lo es. Si esto no trabajase, la secretaria podría generar una tabla con las mismas columnas de numero_Telefono y copiar los datos aquí todos los días. En este caso serían ya sus propios datos, y podría autorizar el acceso.

3.2.4.2.4 Reglas contra disparadores

Son muchas las cosas que se hacen utilizando disparadores que pueden hacerse también utilizando el sistema de las reglas de *Postgres*. Lo que actualmente no se puede implementar a través de reglas son algunos tipos de restricciones. Un *trigger* que se dispare a partir de una *Insert* en una vista puede hacer lo mismo que una regla, situar los datos en cualquier otro lugar y borrar la inserción en una vista. Pero no puede hacer lo mismo en una *update* o una *delete*, porque no hay datos reales en la vista que puedan ser comprobados, y por ello el disparador nunca podría ser llamado, sólo una regla podría realizarlo.

Un disparador se dispara para cada fila afectada. Una regla manipula el árbol de traducción o genera uno adicional. De modo que si se manipulan muchas filas en una instrucción, una regla ordenando una consulta adicional usualmente daría un mejor resultado que un disparador que se llama para cada fila individual y deberá ejecutar sus operaciones muchas veces.

Las reglas sólo serán significativamente más lentas que los disparadores si sus acciones dan como resultado uniones grandes y mal cualificadas, una situación en la que falla el optimizador.

3.2.4.3 Sintaxis de *Create rule*

CREATE RULE name AS ON event TO object [WHERE condition]
DO [INSTEAD] [action | NOTHING]

Descripción

En *Postgres* "el sistema de regla" permite que una acción sea realizada en *updates*, *inserts* o *deletes* en tablas o clases, se utilizan reglas para implementar vistas de tablas. Es pertinente la precaución con reglas de *Sql*. Si el mismo nombre de clase o variable de instancia aparece en el evento, la condición y la parte acción de la regla, son considerados todos diferentes tuplas. De forma más precisa, new y *current* son las únicas tuplas que son compartidas entre cláusulas.

Cuando se elige entre los sistemas de reescritura y reglas de instancia para una aplicación particular de una regla, recuérdese que en el sistema de reescritura, *current* se refiere a la relación y algunos cualificadores mientras que en el sistema de instancias se refiere a una instancia (tupla). Es muy importante notar que el sistema de reescritura nunca detectará ni procesará reglas circulares. El objeto en una regla *Sql* no puede ser una referencia a un *array* y no puede tener parámetros.

3.3 Sistemas de Base de Datos Activas orientadas a objetos

3.3.1 *Chimera*

El sistema *Chimera* combina la tecnología orientada a objeto, deductiva y activa de base de datos. El lenguaje de la regla activa es basado en *Starburst*, con extensiones de orientación a objeto.

3.3.1.1 El modelo de *Chimera*

Es un modelo orientado a objetos, permite manejar objetos complejos, clases, herencia, vistas, restricciones de integridad y disparadores. Los tipos son atómicos o definidos por el usuario. Las clases consisten en atributos y métodos. Los atributos, las vistas, las restricciones de integridad se implementan por medio de expresiones declarativas. Las operaciones y los disparadores se implementan por medio de expresiones procedimentales. Las vistas son definidas fuera de las clases; las restricciones y los disparadores se pueden definir dentro o fuera de las clases.

3.3.1.1.1 Clases

Una clase consiste en los siguientes componentes: atributos, restricciones, operaciones, disparadores, llave (para identificar de forma única al objeto de la clase).

Sintaxis :

```
object_cass_definition ::=
  "define" "object" "class" class_name
  [ ["derived"] "superclasses" class_names]
  ["attributes" attribute_defs]["constraints" constraint_defs]
  ["operations" operation_defs]  ["triggers" trigger_defs]
  ["key" key_def ]["unique" key_defs ]
"end;"
```

3.3.1.1.2 Herencia

Se heredan todas las características: atributos, restricciones, las operaciones, los disparadores. Los atributos y las operaciones pueden ser redefinidos.

Ejemplo:

```
 define object class employee
 superclasses  person
 attributes    emplNr:integer, mgr:employee, salary:integer, dependents:set-of(employee)
 constraints   exceedsMgrSalary
 operations    hire(in Mgr:employee, in StartSalary:integer), fire()
 triggers      adjustSalary
end;
```

3.3.1.1.3 Vistas

Las vistas definen intencionalmente predicados que conectan la información interna o externa. Una vista es similar a la definición de un tipo de registro.

Sintaxis:

```
view::="define" "view" view_name "(" v_field ")"   deductive_rules "end;"
v_field ::= label ":" type
```

Ejemplo:

```
define view marriage(husband:person,wife:person)
 marriage([X,Y]) <- X.spouse=Y, X.sex=male
end;
```

3.3.1.1.4 Restricciones

En *Chimera* se pueden dar dos tipos de Integridad:

- Inmediato: chequea el curso de la transacción; si sucede un evento que altere la transacción se produce un *rollback.*
- Diferido (valor por defecto): chequea antes de hacer un *commit*; si sucede una alteración se produce un *rollback.*
-

Sintaxis:

```
untargeted_constraint ::= "define" [ c_mode ] "constraint" constraint_name "(" c_parameters ")"
deductive_rules "end;" c_parameter ::= label ":" typec_mode ::= "immediate" | "deferred"
```

Ejemplo:

```
define immediate constraint
  illegalMarriage(p1:person,p2: person)
  illegalMarriage([X,Y])<- X.spouse = Y,X.sex=Y.sex
end;
```

3.3.1.2 El Lenguaje de *Chimera*

Es un lenguaje de programación de base de datos usado para implementar clases, restricciones, vistas y disparadores, para la manipulación de datos y transacciones. Se basa en:

- Expresiones declarativas*:* usadas para la escritura de fórmulas, reglas deductivas.
- Las expresiones procedimentales*:* usadas para la parte de acción de la operación y de disparadores, y para las transacciones.

El lenguaje de la regla activa maneja: opción para los tipos de eventos, complejidad del cálculo de eventos, modos de ejecución, une los objetos de la base de datos a los eventos, une los componentes de la regla, maneja prioridades.

3.3.1.3 Sintaxis y semántica de *Chimera Define Trigger*

Los disparadores que se procesan en *Chimera* inmediato o diferido consisten en: reglas accionadas que se ordenan según prioridades, y seleccionar una de las reglas de más alta prioridad.

Sintaxis:
trigger_rule ::= "define" options "trigger" TNAME
"events" events "condition" condition "actions" actions [priority]

Ejemplo :

define trigger *raiseBudget*
events *insert(employee) modify(employee.salary) modify(dept.members) modify(dept.salaryBudget)*
condition *dept(D), integer(I), I=sum(E.salary where employee(E), E in D.members), I $>$ D.salaryBudget* ***actions*** *modify(dept.salaryBudget,D,I) after employee.adjustSalary*

3.3.1.4 Transacciones en *Chimera*

La ejecución de un disparador es permitida entre las líneas de la transacción, pero una línea transacción no puede ser interrumpida por un disparador, incluso si hay reglas accionadas, como operaciones y disparadores, las líneas de transacción son secuencias de comandos o el llamado de operaciones, donde la comunicación es establecida por el paso de parámetro de entrada-salida.

```
create(employee, [firstname: "Piero", lastname:
"Fraternali ", departamento: "CS" ],X), select(Y donde
employee(Y), Y.heads=X.dept), modify(employee.boss, X, Y);
select(Y donde employee(Y), Y.lastname="Fraternali "),
modify(employee.dept, X, "MATH ");
```

3.3.2 *Hipac*

Es una orientada a objetos, creada en 1987 por *Xerox*. *Hipac* fue uno de los primeros sistemas que se crearán con capacidades de proceso activo. *Hipac* fue un proyecto pionero en el área de Sistemas de Base de Datos Activa orientado a objetos. Incluye un poderoso lenguaje de la regla para el modelo de datos orientado a objetos, la semántica de ejecución es flexible, e incluye algunos prototipos experimentales de memoria principal.

El sistema utiliza el modelo estándar orientado a objeto, que se ha extendido para incluir la evaluación de las reglas de ECA. Este sistema se ha diseñado para funcionar en situaciones donde se requiere una respuesta oportuna en situaciones críticas

Los eventos de *Hipac* son muy generales, y puede incluir el lenguaje manipulación de datos, operaciones, esquema de manipulación de operaciones, eventos de transacción, varios tipos de eventos temporales (absoluto, relativo y periódico), y notificaciones externas de los programas de aplicación de base de datos. Estos eventos primitivos pueden combinarse en eventos complejos por la disyunción o sucesión de operadores.

Las condiciones de *Hipac* son consultas en un lenguaje de manipulación de datos orientado a objetos. Las acciones de *Hipac* pueden ser operaciones o mensajes de programas de aplicaciones, como una petición para algún tipo de servicio.

3.3.3 *Ode*

Ode es una base de datos orientada a objetos basada en el paradigma de los objetos de *C++*. *O++* es la interfaz primaria para la base de datos de *Ode*. *O++* amplía *C++* proporcionando los recursos necesarios para implementar las aplicaciones de base de datos, incluyendo las restricciones y los disparadores en los objetos que constituyen los recursos activos para la base de datos.

El diseño de los disparadores y las restricciones en *Ode* incluye:

- Los disparadores y las restricciones se debe especificar declarativamente
- Los disparadores y las restricciones se deben asociar a la definición de la clase para reflejar la orientación a los objetos
- Los disparadores y las restricciones pueden trabajar con el mecanismo de herencia
- Las restricciones aseguran el estado coherente de la base de datos
- Los disparadores no se refieren al estado coherente del objeto
- Las restricciones se aplican en la creación o en la cancelación
- Los disparadores se activa después de la creación del objeto

Ejemplo de Restricciones:

```
class female: public person {
public:
...
constraint: sex == 'f' || sex == 'F';
};
```

Ejemplo de un disparador, considerando la siguiente clase

```
class inventory:
class inventitem: public stockitem {
public:
inventitem (Name iname, double iwt, int xqty, int xconsumption,
double xprice, int xleadtime, Name sname, Addr saddr);
void deposit(int n);
int withdraw(int n); ...
trigger: order(): qty < reorderlevel() ==> place_order(this, eoq());
```

3.3.4 *Polyhedra*

Polyhedra es un servidor de base de datos que se diseño para aplicaciones de negocio y requieren manejo de evento (*event-driven*), maneja respuestas en tiempo real a los cambios de los datos, fiabilidad transaccional y la opción de tolerancia a fallos. Actualmente, la tecnología de base de datos de *Polyhedra* está dirigiéndose a los requisitos de los servidores de alta velocidad en aplicaciones de base de datos-intensivas que van desde la automatización industrial a la infraestructura de las telecomunicaciones inalámbrica, sistemas de administración de red e Internet

Una característica única de *Polyhedra* es que no tiene ninguna limitación para construir sistemas. Si la aplicación tiene un requisito de diseño de base de datos simple o complejo *Polyhedra* permite el desarrollo de aplicaciones con base de datos en tiempo real que procesa ciclos de desarrollo más cortos, con costos bajos y resultados verdaderamente confiables. Características de *Polyhedra*:

A) Funcionamiento de residente en memoria

La base de datos relacional en tiempo real se ejecuta en memoria principal con acceso a disco. La base de datos de memoria principal también proporciona la tolerancia a fallas para la confiabilidad transaccional y el funcionamiento continuo.

Manejo de eventos (*Event-driven)* en todas partes, *Polyhedra* proporciona funcionamiento completo para resolver los requisitos más rigurosos de respuesta de sistemas en tiempo real. Permite la optimización de aplicaciones para que puedan ser puestas en ejecución, que de otra manera serian totalmente imprácticas.

B) Independencia de plataforma

Polyhedra se ejecuta en muchas plataformas incluso en los sistemas operativos de tiempo real más populares, *Windows NT, y UNIX* a través de una variedad amplia de plataformas de dotación física. Un cliente de *Polyhedra* en una plataforma puede comunicarse con un servidor en un tipo diferente de plataforma: la heterogeneidad fue diseñada en el producto.

c) Arquitectura cliente/ servidor

Una arquitectura de cliente/servidor heterogénea proporciona flexibilidad en el diseño de sistemas, y en las aplicaciones distribuidas linealmente-escalables. Un sistema de aplicación distribuido puede correr con los componentes individuales en cualquier combinación de plataformas de hardware; los componentes operan independiente de la plataforma. Apoya las normas como *TCP/IP, SQL, JDBC y OLE/DB* incluso hace globalmente aplicaciones de cliente-servidor distribuidas lo más complejos posible.

d) Servidor de la base de datos del *Web*

Los *browsers* del *Web* pueden dar acceso directo a los datos que se ejecutan en memoria principal. La base de datos de *Polyhedra* incluye clases de *TCP/IP* y de *UDP/IP* para utilizar cualquier método de objeto basado en IP.

Por ejemplo, los métodos del objeto basado en *HTTP* pueden utilizar las páginas dinámicas creadas en el *Web*, así que *Polyhedra* puede ser utilizada como un servidor *Web* y base de datos. El significado real de esto es la capacidad de hacer una fuente de información en línea para cualquier aplicación de *Polyhedra* disponible para los funcionamientos de alta velocidad de operaciones de lectura/escritura de los clientes browsers estándares del *Web*. Como servidor *Web*, puede entregar las paginas usando la información actualizada de una variedad de fuentes de información que se ejecutan en el proceso mismo de la base de datos y accesible a otro objeto de información guardado en la base de datos.

e) Base de Datos Activa

La naturaleza activa de *Polyhedra* permite código completo en la aplicación para residir y ejecutarse dentro de la propia base de datos, proporcionando flexibilidad y ejecución sobre los sistemas de base de datos pasivas.

f) Consultas activas

Polyhedra incorpora un concepto único llamado consultas activas (*Active Queries*) que permiten declaraciones de *SQL* y de objetos. Una consulta activa retorna automáticamente información acerca de los cambios que afectan un conjunto de resultado. Esto permite que los clientes del servidor de *Polyhedra* reciban continuamente consultas, incluyendo todos los cambios individuales del atributo de cualquier registro afectado.

g) Base de datos objeto/relacional

Polyhedra une las tecnologías orientadas a objetos y relacionales de las bases de datos. La base de datos amplia *SQL* orientado a objetos para apoyar (herencia, encapsulación, polimorfismo, etc.) con un lenguaje de control para diseñar los métodos de los objetos.

La interfaz relacional de SQL proporciona a los diseñadores el mejor de ambos paradigmas que tiene una base de datos relacional y de SQL estándar que utiliza funciones orientadas a objetos.

d) Un tiempo de desarrollo más rápido

Las interfaces de desarrollo para *Polyhedra* son impresionantes y permiten el uso de tecnología disponible de las herramientas de desarrollo. Además de *ODBC* y del *SQL* estándar, incluye un lenguaje de control (*CL*) usado para implementar definiciones de clase, encapsulación de datos, herencia y polimorfismo de gran alcance para fines generales, orientados a objetos. *C* y *C++*. *APIs* para el acceso del cliente, programar las comunicaciones, junto con un conjunto de bibliotecas pre-construidas de clase.

3.3.4.1 Evitar la necesidad de mecanismos de concurrencia en base de datos

Los sistemas de administración de base de datos tradicionales proporcionan un mecanismo de bloqueo para la ejecución de transacción en paralelo para permitir que los programadores de aplicación realicen las actualizaciones y que mantengan una política de integridad a nivel de aplicación, mientras que no se bloquea y sigue la actividad de la base de datos. Se describe cómo los Sistemas de Administración de Base de Datos Activa puede evitar la necesidad de tales mecanismos, con las ventajas subsecuentes en funcionamiento del sistema y la simplicidad de la implementación.

3.3.4.1.1 Introducción

Uno de los problemas principales del diseño de base de datos es la especificación y la aplicación de una política de integridad a nivel de la aplicación. La mayoría de los sistemas de administración de la base de datos, (DBMS) harán cumplir un conjunto básico de políticas de integridad.

Por ejemplo, los sistemas de administración de la base de datos relacionales garantizarán, entre otras cosas, que las llaves primarias del atributo identifican únicamente al registro, que cada valor del atributo es del tipo correcto - pero en general, un diseñador de la base de datos deseará ir más lejos que esto.

Las técnicas tales como normalización ayudan, pero no son suficientes, así que con el DBMS tradicional, el diseñador de la base de datos especifica no solamente el esquema de la base de datos sino que también un conjunto de reglas para cada programa de la aplicación se codifican los procesos del cliente que hacen uso de la base de datos.

Para ilustrar esto, considere un área de aplicación simple: un registro de los nacimientos, muertes y matrimonios, puede ser implementados usando bases de datos relacionales. Esto podría consistir en dos tablas, una con información sobre la persona - sus nombres y su fecha de nacimiento y muerte - y otra que lleva la información sobre sus relaciones por medio del matrimonio.

Para capturar el hecho que en la mayoría de los países sea ilegal ser casado con más de una persona al mismo tiempo, el esquema de la base de datos es suplido por las guías de consulta para escribir los programas de aplicación, siempre que la base de datos sea actualizada esta regla debe de mantenerse, es decir que no haya una persona casada con mas de una persona. Así, que al actualizar la base de datos para registrar una muerte, es necesario cancelar cualquier unión relacionada, y al registrar una nueva unión en primer lugar se tiene que asegurar que los registros existen para la gente implicada, de que ninguna de las dos personas este marcado como muertas y que no se encuentren casados con otra persona.

En esta etapa se presentan los problemas de la ejecución simultánea (concurrencia): si las consultas de la base de datos y la actualización de la base de datos se realizan como operaciones independientes, entonces hay la posibilidad (quizá solamente leve, pero bastante verdadera) que otra aplicación actualice la base de datos entre las consultas y la actualización, de tal manera de invalidar las revisiones anteriores; por ejemplo, la operación de introducción de la muerte de uno de los implicados.

Tradicionalmente, la aplicación de la base de datos bloquee a la base de datos (de modo que ningún otro cliente pueda interferir con los datos de una manera tal que el cambio dé lugar a una violación de integridad); por razones del funcionamiento, se ofrece granularidad, de modo que mientras que una aplicación está ocupada otras aplicaciones puedan realizar otra tarea y las operaciones no entran en conflicto. Con este beneficio, el proceso de registrar un matrimonio puede ser:

- Indicar a la base de datos que se va a comenzar una nueva transacción;
- Buscar el registro de cada una de las partes implicadas, usando una primitiva del DBMS que indique que el registro - fue encontrado - y debe ser bloqueado;
- Si la búsqueda no tiene éxito, se señala el fin de la transacción, se revisa la bandera que indica que la persona esta muerta. Si es así se cancele la transacción, que cancelará automáticamente el bloqueo establecido previamente;
- Si se asume que la búsqueda anterior tuvo éxito y se verifica que la persona está viva, se repite la búsqueda y se aplica el procedimiento para verificar que la otra persona esta casada, se cancele la transacción como antes si hay problemas - en caso, que ambos registros estén bloqueados, la cancelación se hará automáticamente;
- Se controla si hay uniones existentes, y se bloquea simultáneamente la tabla de matrimonios de modo que ningún otro proceso del cliente pueda agregar un registro

- Finalmente, se controla los resultados de las consultas de la unión, y se cancela la transacción entera o se inserta un nuevo registro de la unión y se señale que esto termina la transacción: en cualquier caso, los bloqueos excepcionales serán cancelados.

En la práctica, algunos de estos pasos se pueden dar juntos, pero como puede ser visto, la operación no es simple, y requieren un número de interacciones con la base de datos; el peor de todos quizás, le da la responsabilidad al programador para implementar correctamente las reglas especificadas por el diseñador de la base de datos.

También complica perceptiblemente el diseño y la implementación del sistema de administración de la base de datos, que tiene que poder hacer frente a las transacciones simultáneas múltiples, cada uno de las cuales se puede cancelar en cualquier momento, requiriendo la cancelación de cualquier bloqueo y de deshacer cualquiera de los cambios realizados hasta ese momento dentro de la transacción.

El DBMS también tiene que controlar si hay requisitos del bloqueo que están en conflicto, manteniendo o cancelando las transacciones donde sea necesario para evitar o resolver conflictos. Finalmente, los desarrolladores de sistema de administración de la base de datos tienen que proporcionar un conjunto de mecanismos de bloqueo, para permitir que el diseñador de la base de datos evite restricciones innecesarias en la clase de transacciones que se puedan realizar en paralelo.

Particularmente en sistemas de tiempo real donde es crítico el funcionamiento, los recursos son limitados pero las operaciones típicas realizadas no son complejas, las técnicas alternativas son necesarias. Una técnica, llamada Bases de Datos Activas, se demostrará cómo se puede reducir la necesidad de transacciones simultáneas, y se describe un DBMS activo disponible en el comercio que no tenga concurrencia de transacción.

3.3.4.1.2 Bases de Datos Activas

La definición de un DBMS activo usado aquí es un DBMS que proporciona al diseñador de la base de datos la oportunidad de asociar código a un esquema que se accione automáticamente cuando ocurren los cambios.

Yendo de nuevo al ejemplo anterior, la operación de registrar un nuevo matrimonio se puede simplificar enormemente. La regla que se seguirá se reduce a solo un paso: inserte un nuevo registro de matrimonio. Todo el resto de las verificaciones es realizado automáticamente, en parte por el DBMS como resultado del diseño cuidadoso del esquema, y en parte por el código asociado al esquema por el diseñador de la base de datos.

La responsabilidad de especificar la política de la integridad no es solo del diseñador de la base de datos, y parte de este mecanismo esta integrado en el esquema de la base de datos. Sin embargo para justificar la especificación de las reglas recuerde que parte esta en el mecanismo de la base de datos, el diseñador de la base de datos codifica estas reglas y las une a la base de datos.

Se observa que este recurso no restringe al desarrollado de la aplicación, elimina simplemente algo de la responsabilidad y reduce el número de las interacciones de la base de datos que se requieren; incluso si la aplicación realiza sus propios chequeos, no existe el bloqueo necesario puesto que los chequeos finales son realizados por el código activo como parte de la operación de insertar. Así por lo que la integridad total no importa si la aplicación realiza un número de pasos:

- Se consulte la base de datos para obtener los registros de las personas, lo cual puede ser expresados en SQL como:

```
select id,bithdata,birthplace,dathdate
from person
where name='John Doe' or name='Anne Smith';
```

- Se analiza la respuesta para determinar si *John Doe* y *Alice Smith* están vivos. Nótese que esta fase no involucra ninguna interacción con la base de datos, los DBMS no necesitan tener cualquier recolección de consultas, y la base de datos puede cambiar durante esta fase;
- Finalmente, se agrega el registro a la base de datos para el nuevo matrimonio, por ejemplo:

insert into marriage (locationcode, id, husband, wife, mdate)
values (12345, 1289, 568746282, 834772012, now ());

Otros beneficios de este acercamiento permiten que se cambien las reglas de integridad sin la necesidad de volver a escribir las aplicaciones existentes. Por ejemplo, es obvio que la pudiera agregar chequeos como si las personas que se van a casar son de edad legal y - donde es un requisito legal – diferencia de género.

Nótese que no todos los DBMS's activos ofrecen las misma facilidades, y así algunos pueden permitirle al diseñador de la base de datos aplicar reglas más estrictas que otros. Por ejemplo, un DBMS que permite que se ejecute código después de todo los cambios que el cliente a solicitado podría significar que el código activo realmente no trabaja como se esperó, si se hacen cambios relacionados a la base de datos dentro de una sola transacción - si en el ejemplo un lote de cambios entrara al final de un mes e incluye el matrimonio de una persona y también un registro de su muerte, el código activo podría rechazar la operación porque el matrimonio se refiere a alguien que ya es marcado como muerto. Obviamente, se debe tener más cuidado en el código (como verificar la fecha de muerte contra la fecha de matrimonio, la fecha de muerte debe especificarse, en lugar de simplemente insistir que la fecha de muerte es nula) habría en este caso un problema, pero en general los diseñadores de base de datos lo pueden hacer más fácil sí:

- El código activo apropiado es ejecutado por el DBMS después de cada comando individual de la operación requerida por el cliente;

- El DBMS permite la granularidad para el código activo que se une a la base de datos, por ejemplo permitiendo activar diferente código dependiendo cuales atributos sean alterados.

3.3.4.2 Desnormalización

Fue mencionado antes que es común normalizar bases de datos relacionales para mejorar la integridad de la base de datos. En una base de datos normalizada, el esquema se diseña para que un elemento de información se mantenga en un único lugar, y que esa información no se pueda deducir de otra información de la base de datos.

Así, el principio de normalización diría que no se tiene una bandera en la tabla de *person* para decir que si dos personas están casados, puesto que una consulta diseñada apropiadamente en la tabla *marriage* mostrara este hecho; esto ayuda a la integridad global de la base de datos, puesto que si no hay ninguna bandera no hay ningún riesgo que su valor difiere de la consulta. Sin embargo, la normalización puede ser cara en funcionamiento, puesto que para determinar una consulta, que liste a los hombres casados requieren más de una tabla.

```
select p.id, p.name rom  person p, marriage m  where m.husband = p.ID and  m.enddate is null;
```

Y una consulta para encontrar a los solteros elegibles entre la edad de 18 y 30 se vuelve compleja. Si se usa un DBMS activo, sin embargo, se vuelve posible agregar atributos extras para simplificar las consultas (y/o para el propio código activo) sin arriesgar la integridad. Así, suponga que el código de la mantiene un atributo 'married' en el registro de tipo de la tabla *person*, entonces la consulta de los solteros elegible se vuelve:

```
select id, name from person where born + years (25)  > now ()
and born + years(18) < now ()       and   married is null and died is null;
```

3.3.4.3 Ejemplo

Usando Bases de Datos Activas se le permite al diseñador de la base de datos manejar la de integridad de las aplicaciones del cliente en la base de datos; con disparadores con granularidad, la necesidad por bloquear y trabajar las transacciones concurrentes puede diseñarse fuera de la aplicación. Las bases de datos de tiempo real pueden contar con esto y pueden evitar la necesidad de proporcionar cualquiera bloqueo para el soporte de la ejecución en paralelo (concurrencia), especialmente si un alto nivel de control de concurrencia puede proporcionarse para proveer la naturaleza activa de la base de datos.

3.3.4.3.1 La base de datos *marriage*

Se muestra a continuación el esquema y el código activo para una base de datos de matrimonio. El esquema esta escrito en el lenguaje de *SQL*, y el código activo se escribe en *CL* que programa el lenguaje de desarrolló para el usó del sistema de administración de *Polyhedra*.

3.3.4.3.2 El esquema

El ejemplo hace use de *'crate schema'*, permitiendo introducir un grupo de tablas:

```
create schema
create table person
   (persistent, id  integer primary key,  name      char not null,
    born   datetime, died  datetime, ismale bool,
    married   integer references marriage
   )
create table marriage
   (persistent, id  integer primary key,
    wife  nteger not null references person,
    husband   integer not null references person,
    started   datetime not null,location  char, ended  datetime,
    whyended  char
  );
```

3.3.4.3.3 El código activo

El mecanismo del código activo en *Polyhedra* le permite al diseñador de la base de datos unir fragmento de código, o métodos, escrito en un lenguaje codificando especial llamado *CL* con varias tablas en la base de datos. *CL* es bastante autoexplicativo.

'--' indica que el resto de la línea es un comentario,
' \ ' backslash es el carácter de continuación de línea,
' & ' ampersand concatena strings, y
' &&' doble-ampersand concatena un carácter de espacio entre los strings a encadenarse.

```
script marriage

--el 'script' CL introduce un  grupo de métodos para ser unidos al nombre de la tabla '
-- end script' indica el fin del grupo de métodos.
On  create
--en Polyhedra, 'on create' son métodos que  se activan cuando los objetos se crean en la base de datos,
--esto significa que cuando se usa una sentencia de SQL INSERT o por código de CL se ejecutan
varias
if started  > now () then
        abort transaction \ start date cannot be in the future."
else if exists ended then
        --el final de los campos es no nulo – así este registro no afecta el  estado de quién se casa.
        --Por consiguiente, se puede ignorar el registro
        exit
else if exists married of husband then
         abort transaction
         \  name of husband  && \ "is already married to"
        && \  name of wife of married of husband
 else if exists married of wife then
        abort transaction \
        name of wife  && \   "is already married to"  && \
        name of husband of married of wife
else if not ismale of husband then
         abort transaction \
        "husband should be male,"  && \  name of husband  && "is not."
else if ismale of wife then
        abort transaction \
        "wife should be female,"  && \ name of wife  && "is not."
else if exists died of husband or \
exists died of wife then
        abort transaction \
        "both spouses must be alive to marry."
end if
--se actualiza los otros registros en la base de datos para mantener la consistencia
        set married of husband to me
        set married of wife to me
end create
```

```
on  set husband
 -- se activa cuando un  atributo se altera (por un SQL UPDATE, o un conjunto de   declaración CL);
 -- aquí, se puede prohibir el cambio.
           abort transaction \
          "cannot alter the husband of a marriage"
 end set husband
 on  set wife
           abort transaction \
           "cannot alter the wife of a marriage"
 end set wife
on  set started
          abort transaction \
          "cannot alter the start date of a marriage"
 end set started
 on  set ended
        --marca un matrimonio  colocando este atributo como no-nulo
if not exists ended then
          --la fecha final se alteró de no-nulo a nulo: error en la aplicación
          abort transaction \
          "cannot unset end date of a marriage"
else if ended  < started then
          abort transaction \
           "a marriage cannot end before it begins!"
else if married of husband = me then
          --el matrimonio fue actualizado;
          set married of husband to null
          set married of wife    to null
else
          abort transaction \
           "cannot alter end date of a marriage"
 end if
 end set ended
 end script
script person
on create
 if born  > now () then
          abort transaction \
          "date of birth must not be in the future."
 else if exists died then
          --el registro se ha creado con la fecha de nacimiento y la fecha de muerte - qué está bien,
          --con tal de que ambas fechas estén en el pasado y  en un orden sensato...
if died  < born then
          abort transaction \
              "cannot die before being born"
else if died  > now () then
           abort transaction \
           "cannot know when"  && name  && \  "will die"
 end if
 end if
 end create
on set died
if not exists died then
          abort transaction \
          "cannot raise the dead"
```

```
else if died  < born then
        abort transaction \
        "date of death must not precede birth"
else if died  > now () then
        abort transaction \
         "cannot know when"  &  &  name  &  &  \ "will die" \
else if exists married then
        set whyended of married to "death of"  && name
set ended
        of married to now ()
end if
end set died
on  set ismale
    --Se podría generar cambios, o reglas, si la persona está actualmente casada, o el matrimonio este
    --cancelado
if exists married then
        set whyended of married to \
        name  && "had a sex change"
set ended of married to now ()
end if
end set ismale
end script
```

Además se le puede agregar código, por ejemplo, uno puede agregar código para asegurar que el atributo *married* de la tabla *person* hace referencia a un registro de *marriage.* Además, los mecanismos de seguridad pueden restringir quienes pueden hacer alteraciones, puede prepararse para que ninguno pueda alterar directamente atributos (como married) en cualquier caso las únicas alteraciones posibles son hechos por código de *CL.*

3.3.4.3.4 Ejemplo de *SQL*

Las declaraciones de *SQL* siguientes ilustran cómo la base de datos puede multiplicarse y puede ser consultada por clientes.

- Primero, se crean algunos registros en la tabla *person*:

```
insert into person (id, name, born, ismale) values (1, 'Adam', date ('01-jan-1951'), true);
insert into person (id, name, born, ismale) values (3, 'Ben', date ('01-dec-1953'), true);
insert into person (id, name, born, ismale) values (5, 'Chris', date ('01-nov-1955'), true);
insert into person (id, name, born, ismale) values (7, 'David', date ('01-oct-1962'), true);
insert into person (id, name, born, ismale) values (9, 'Edward', date ('01-sep-1959'), true);
insert into person (id, name, born, ismale) values(11, 'George', date ('01-aug-1961'), true);
insert into person (id, name, born, ismale) values (13, 'Hugh', date ('01-jul-1963'), true);
insert into person (id, name, born, ismale) values (15, 'Ian', date ('01-jun-1965'), true);
insert into person (id, name, born, ismale) values (17, 'John', date ('01-may-1967'), true);
insert into person (id, name, born, ismale) values (19, 'Keith', date ('01-apr-1969'), true);
insert into person (id, name, born, ismale) values (100, 'Anne', date ('01-jan-1951'), false);
```

```
insert into person (id, name, born, ismale) values (102, 'Betty', date ('02-feb-1952'), false);
insert into person (id, name, born, ismale) values (104, 'Carol', date ('03-mar-1953'), false);
insert into person (id, name, born, ismale) values (106, 'Doris', date ('04-apr-1954'), false);
insert into person (id, name, born, ismale) values (108, 'Emma', date ('05-may-1955'), false);
insert into person (id, name, born, ismale) values(110, 'Fiona', date ('06-jun-1956'), false);
insert into person (id, name, born, ismale) values(112, 'Gina', date ('07-jul-1957'), false);
insert into person (id, name, born, ismale) values(114, 'Helen', date ('08-aug-1958'), false);
insert into person (id, name, born, ismale) values(116, 'Ina', date ('09-sep-1959'), false);
insert into person (id, name, born, ismale) values (118, 'Jane', date ('10-oct-1960'), false);
commit;
```

- se crean algunos registros válidos en la tabla *marriage*...

```
insert into marriage (id, husband,wife,started, location)
values (1, 1, 100, date ('11-jul-1970'), 'London');
insert into marriage (id, husband,wife,started, location)
values (2, 3, 102, date ('21-jun-1972'), 'London');
insert into marriage (id, husband,wife,started, location)
values (3, 5, 104, date ('07-may-1971'), 'Cambridge');
insert into marriage (id, husband,wife,started, location)
values (4, 7, 106, date ('11-jul-1970'), 'Edinburgh');
insert into marriage (id, husband,wife,started, location)
values (5, 9, 108, date ('30-Dec-1977'), 'Edinburgh');
insert into marriage (id, husband,wife,started, location)
values (6, 11, 110, date ('10-Mar-1978'), 'Edinburgh');
commit;
```

- Se ingresa un divorcio y unas segundas nupcias:

```
update marriage set ended=date('03-Apr-1983'), whyended='Divorce'  where id = 3;
insert into marriage (id, husband,wife,started, location)
values (7, 5, 112, date ('15-jul-1983'), 'London');
 commit;
```

- Se ingresa una muerte y unas segundas nupcias:

```
update person set died=date('09-Feb-1985') where id = 1;
insert into marriage (id, husband,wife,started, location)
values (9, 13, 100, date ('15-May-1986'), 'Cardiff');
commit;
```

- Se ingresan divorcios:

```
update marriage set ended=date('24-Aug-1986'), whyended='Divorce' where id = 8;
update marriage set ended=date('18-Sep-1986'), whyended='Divorce'  where id = 6;
update marriage set ended=date('02-Oct-1986'), whyended='Divorce'  where id = 7;
commit;
```

- y al final se ingresan 4 bodas y un funeral.

```
insert into marriage (id, husband,wife,started, location)
values (10, 11, 112, date ('10-May-1990'), 'London');
insert into marriage (id, husband,wife,started, location)
values (11, 15, 114, date ('11-Jun-1990'), 'London');
insert into marriage (id, husband,wife,started, location)
values (12, 17, 116, date ('12-Jul-1990'), 'London');
```

```
insert into marriage (id, husband,wife,started, location)
values (13, 19, 104, date ('13-Aug-1990'),  London');
update person set died=date('09-Feb-1985') where id = 106;
commit;
```

Se puede observar algunas consultas simples y los resultados que se obtendrían por medio de *Sql*.

- Listar a las personas casadas:

Esto podría ser tan simple como *'select id, name from person where married is not null'* o se podría hacer que se mostrara la edad de la persona en el momento del matrimonio. Esta última petición requiere información de las tablas *person* y *marriage*, pero esta consulta usa una llave extranjera que debe hacer referencia a otra tabla con esto se puede realizarse eficazmente la consulta.

```
select p.id id, p.name name, ismale, the_year(m.started) married,the_year(m.started-p.born) aged
from   person p, marriage m where  p.married=m.id;
```

Id	Name i	smale	Married	Aged
3	Ben	TRUE	1972	19
9	Edward	TRUE	1977	19
11	George	TRUE	1990	29
13	Hugh	TRUE	1986	23
15	Ian	TRUE	1990	26
17	John	TRUE	1990	24
19	Keith	TRUE	1900	22
100	Anne	FALSE	1986	36
102	Betty	FALSE	1972	21
104	Carol	FALSE	1990	38
108	Emma	FALSE	1977	23
112	Gina	FALSE	1990	33
114	Helen	FALSE	1990	32
116	Ina	FALSE	1990	31

- listar a las personas fallecidas:

```
select id, name dead, ismale, the_year(died-born) aged  from   person where  died is not null;
```

Id	Name	Ismale	Aged
1	Adams	TRUE	35
106	Doris	FALSE	31

- Listar a las personas solteras:

```
select id, name, ismale, the_year(now()-born) aged
from   person where  married is null and died is null;
```

Id	Name	Ismale	Aged
5	Chris	TRUE	44
7	David	TRUE	37
110	Fiona	FALSE	43
118	Jane	FALSE	39

- Listar información sobre los matrimonios:

Para desplegar los nombres de las personas que están casados se realiza una unión (join) a 3 tablas (*marriage x person x person*), pero como en el ejemplo anterior se están utilizando llaves extranjeras para seleccionar los registros en otra tabla y así que puede realizarse eficazmente.

select the_year(m.started) year, location,h.name husband, the_year(m.started-h.born) age_h, w.name wife, the_year(m.started-w.born) age_w, the_year(m.ended) ended, whyended from marriage m, person h, person w where m.husband=h.id and m.wife=w.id order by m.started;

Year	Location	husband	Age_h	Wife	Age_w	ended	Whyended
1970	Edinburgh	David	8	Doris	17	1198	Death of Doris
1970	London	Adam	20	Anne	20	1998	Death of Doris
1971	Cambridge	Chris	16	Carol	19	1983	Divorce
1972	London	Ben	19	Betty	21	NULL	NULL
1977	Edinburgh	Edward	19	Emma	23	NULL	NULL
1978	Edinburgh	George	17	Fiona	22	1986	Divorce
1983	London	Chris	28	Gina	27	1986	Divorce
1986	Cardiff	Hugh	23	Anne	36	NULL	NULL
1990	London	George	29	Gina	33	NULL	NULL
1990	London	Ian	26	Helen	32	NULL	NULL
1990	London	John	24	Ina	31	NULL	NULL
1990	London	Keith	22	Carol	38	NULL	NULL

Se observa los resultados que se produjeron cuando se hicieron varios cambios en la base de datos.

```
SQL > update person set died=NULL where id = 1;
SQL > commit;
SQL > Error: cannot raise the dead
SQL > insert into marriage (id, husband,wife,started, location) values (100, 1, 110, date ('10-Aug-1995'), 'London');
SQL > commit; Error: both spouses must be alive to marry.
SQL > insert into marriage (id, husband,wife,started, location)  values (101, 110, 5, date ('10-Aug-1995'), 'London');
SQL > commit; Error: husband should be male, Fiona is not.
SQL > insert into marriage (id, husband,wife,started, location)  values (102, 5, 110, date ('10-Aug-2095'), 'London');
SQL > commit; Error: start date cannot be in the future.
SQL > insert into marriage (id, husband,wife,started, location) values (102, 19, 110, date ('10-Aug-1995'), 'London');
SQL > commit; Error: Keith is already married to Carol
SQL > update marriage set husband = 5 where husband = 19;
SQL > commit; Error: cannot alter the husband of a marriage
SQL > update marriage set ended = date ('01-Jan-1930') where husband = 19;
SQL > commit; Error: a marriage cannot end before it begins!
SQL > update person set died =  date ('01-Jan-2130') where id = 19;
SQL > commit; Error: cannot know when Keith will die
```

4. METODOLOGÍA DE IDEA

4.1 Proyecto IDEA

Un proyecto financiado por la Comunidad Europea, inició en junio de 1992 y terminó en abril de 1997. El objetivo fue desarrollar un prototipo de sistemas activos deductivo orientado a objetos y proporcionar la metodología y las herramientas para extender el uso de la tecnología activa de *DOOD* y de integrar de forma coherente los conceptos, metodología y herramientas necesarios para diseñar y desarrollar aplicaciones de bases de datos.

Dentro de IDEA, *SEMA Group* se encarga de validar y evaluar la tecnología de IDEA para aplicaciones de *Workflow*. Los participantes en IDEA son: *Bull* (Francia - Coordinador), *BIM* (Bélgica), *ECRC* (Alemania), *INRIA* (Francia), *KPN* (Holanda), Politécnico di Milano (Italia), *SEMA* (España), TXT (Italia), Universidad *Bonn* (Alemania). La tecnología de IDEA incluye Orientación a Objetos, reglas deductivas y reglas activas.

IDEA percibió las siguientes necesidades y problemas:

- La necesidad de aumentar la oferta de sistemas y de productos tales como: sistemas de administración de bases de datos relacionales activos, sistemas de administración de bases de datos orientadas a objetos activos, sistemas deductivos orientados a objetos.

- Deseo de explorar la tecnología extendida de base de datos a su capacidad máxima. Divulgación del conocimiento académico en el desarrollo basado en reglas de aplicación.

Que ofrece la metodología IDEA:

- Primitivas de esquema
  Tipos, clases
- Conocimiento declarativo
  -Integridad incorporada: llave, cardinalidad, integridad referencial
  -Restricciones genéricas: estática/dinámica, operacional/transaccional
  -Datos derivados: atributos derivados, vistas
- Conocimiento procesal
  -Reglas activas para establecer restricciones y la derivación de los datos
  -Reglas para asociar comportamiento dinámico
  -Reglas de negocio
- Proceso
  - Análisis y atención en aspectos relacionados con base de datos
  - Diseño
    -Guías de consulta para la transición de lo informal a lo formal
    -Clases, generalización, diseño de relaciones
    -Diseño de operación
    -Diseño de restricciones
    *Diseño de las reglas de negocio
  - Prototipos y verificación
    -Reglas deductivas: estratificación
    *Reglas activas: modularización, estratificación

- Implementación

    -DBMSs activos relacionales: *Sql3*, *Orácle*

    -*Oo* activo: *Ode, Naos, Chimera*

    -DOOD: ADS

Cuando se podría utilizar la metodología IDEA:

- Para usar las características avanzadas de la tecnología de base de datos:

    -Objetos

    -Reglas

- Para reutilizar disciplinas de software:

    -Lenguajes Orientados a objetos

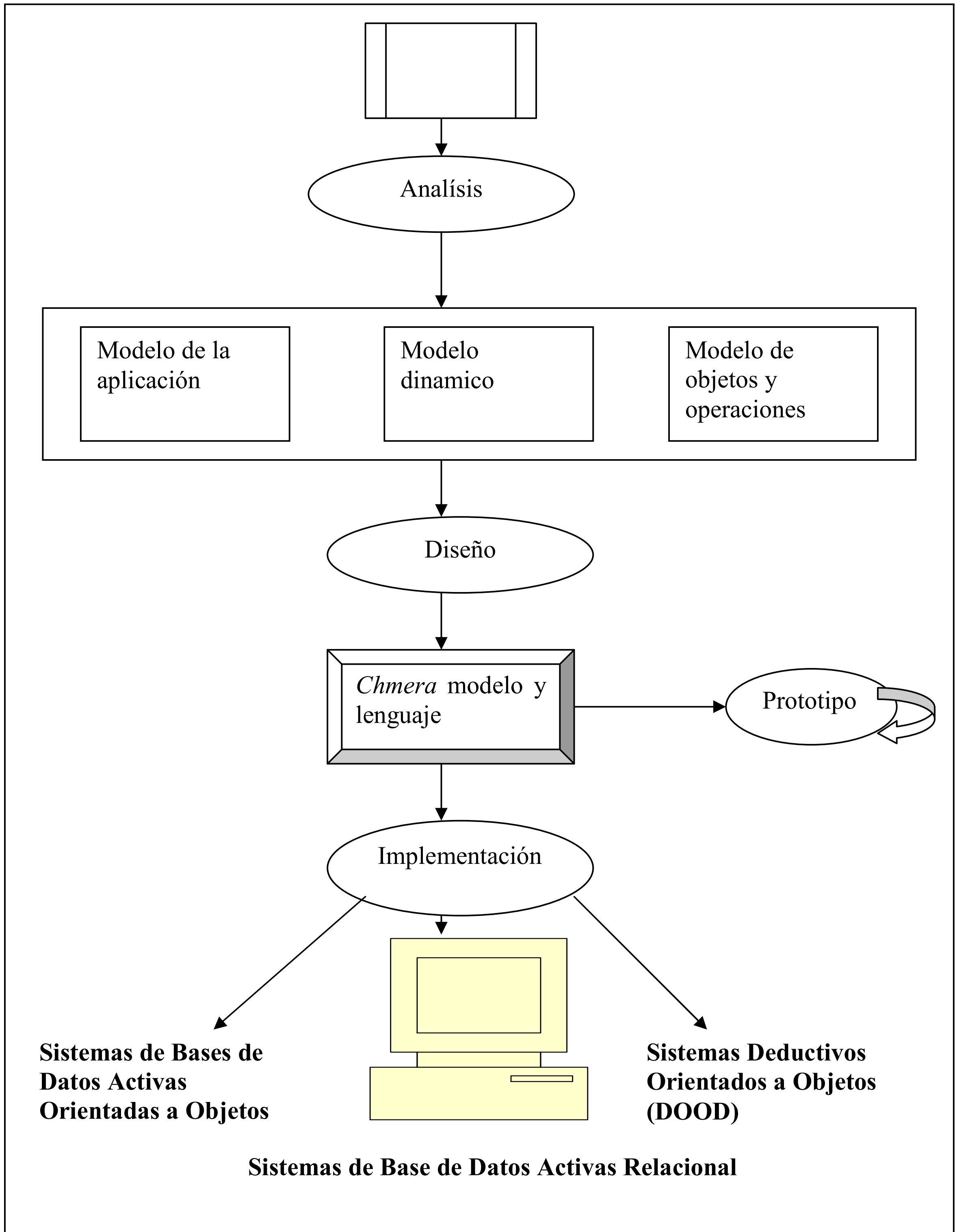
    -Sistemas expertos

- Generación de las reglas activas para las aplicaciones específicas (*workflow*)

4.2 Estructura del proceso IDEA

El proceso de IDEA inicia con los requerimientos, apartir de ellos se realiza el análisis, el diseño, el prototipo, usa el modelo y el lenguaje de *Chimera*, luego realiza la implementación ya sea de Sistemas de Base de Datos Activas Orientadas a Objetos, sistemas deductivos orientados a objetos, sistemas activos relacionales. En la figura 7 se muestra el diagrama de cómo esta compuesta la estructura de ***IDEA***.

Figura 7. Estructura del proceso IDEA



4.3 Diseño de la regla activa

Es la contribución más original de la metodología de IDEA, el diseñar reglas individuales que responden a los requisitos del análisis seguido por el prototipo de la regla activa que se centra en el comportamiento colectivo de un conjunto de reglas Las reglas activas se usan para el mantenimiento de la integridad, con los objetos dinámicos, para el proceso del negocio.

Clasificación de la regla activa:

Reglas internas: soportan características internas de bases de datos, como las restricciones que controlan la integridad, mantenimiento de datos derivados, y réplica de los datos. Aplicaciones internas, como el mantenimiento de versiones, y la administración de la seguridad.

Reglas internas extendidas: soporta aplicaciones específicas construidas alrededor del núcleo (*kernel*) de la base de datos, tal como algunos sistemas de administración de procesos de negocio (*workflow*).

Reglas externas: provienen de los requisitos de una o más aplicaciones; como las alertas, cuando advierten simplemente al usuario sin cambiar el contenido de la base de datos; o reglas de negocio, cuando realizan una parte del cómputo del negocio que sería normalmente aplicaciones específicas.

El diseño activo de la regla se centra en reglas internas, extendidas, y externas y define los componentes principales de la regla (evento, condición, acción, prioridad), y sus opciones semánticas (información transferida por parte del evento a la condición, y de la condición a la acción, el modo de acoplamiento del evento-condición, modo de finalización del evento).

Reglas activas para el mantenimiento de la integridad pueden ser diseñadas:

1. Como reglas deductivas declarativas que definen un predicado que se puede utilizar para consultar violaciones de integridad.
2. Como reglas de interrupción, por ejemplo, disparadores activados por cualquier evento que viola la integridad, mientras que la condición verifica la ocurrencia de violaciones de integridad, y que acción podría realizar *rollback* si una violación se detecta realmente.
3. Como reglas de mantenimiento de integridad, por ejemplo, disparadores que tienen los mismos eventos y la misma condición como reglas de interrupción, pero una acción que repara las violaciones de integridad.

Estas alternativas se aplican al diseño de formato fijo y las restricciones genéricas. IDEA aconseja: Comenzar con 1; Luego con 2 ó 3 si el sistema no utiliza chequeos declarativos de integridad; prefiera 3 cuando se tienen transacciones largas y conozca como reparar las violaciones de integridad. El paso a partir del 1 al 3 puede (parcialmente) ser automatizado. Ejemplos de las reglas activas para el mantenimiento de la integridad:

Restricción: un empleado no debe ganar más que su administrador. Formulación declarativa por medio de una regla deductiva:

```
add constraint mgrSalary() for employee as
 mgrSalary(Self)<-Self.salary > Self.manager.salary
end;
```

Reglas de interrupción (*Abort rule*)

```
define deferred trigger mgrSalary for employee
events      create,modify(salary),modify(manager)
condition   Self.salary > Self.manager.salary
action      rollback
end;
```

```
define deferred trigger mgrSalary for employee
events    create, modify(salary), modify(manager)
condition Self.salary > Self.manager.salary
action    modify(salary, Self.manager, Self.salary)
end;
```

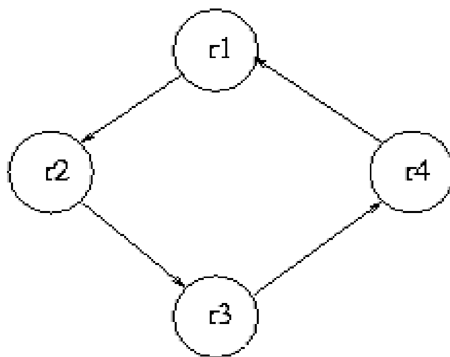
4.4 Prototipo de regla activa

El objetivo es la evaluación del comportamiento colectivo de un conjunto de reglas. Como ayuda al paso subsecuente de la implementación, el prototipo incluye técnicas para asociar las reglas deductivas (utilizadas por pocos sistemas comerciales) en reglas activas

4.4.1 Propiedades de la ejecución de la regla activa

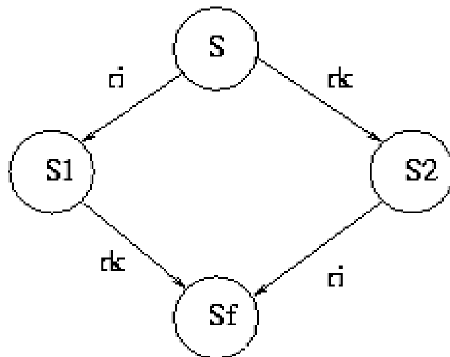
Fin de la regla o terminación: Para cualquier transacción legal, la ejecución subsecuente de la regla termina. Ver figura 8.

Figura 8. Fin de la ejecución de la regla



Confluencia: Para cualquier transacción legal, la ejecución subsecuente de la regla termina en el mismo estado final. Ver figura 9.

Figura 9. Confluencia de la ejecución de la regla



Funcionamiento observable idéntico: Para cualquier transacción legal, la ejecución subsecuente de la regla es confluente y produce la misma secuencia de la salida. A continuación se describen aspectos importantes sobre las reglas activas:

A) Análisis de terminación de la regla

- Una tiene comportamiento de no terminar si existe por lo menos una transacción que produce un proceso de regla que no ha terminado.

- Gráfico de un *trigger* (GT):
  - Grafo dirigido ***{V,E}***
  - Un nodo ***{vi ε V}*** corresponde a una regla ***{ri ε R}***
  - Un arco ***{rj,rk ε E}*** significa que la acción de la regla ***rj*** generada los eventos cuando la regla es disparada ***rk***
  - El grafo del *trigger* es acíclico esto implica la ausencia de no terminar.

B) Un análisis mejor de la finalización de la regla

- Problemas: el número de arcos en el GT puede ser muy grande.
- Un ciclo en el GT es necesario, pero no suficiente, para que la condición termine

La meta es reducir el gráfico del *trigger* eliminando los arcos que no pueden producir activaciones en tiempo de corrida, usando la técnica de encontrar contradicciones entre las actualizaciones de una regla y la condición de otra

C) Ejemplo de la simplificación del TG

define trigger cancelWire for class tube
events delete
condition wire(X), X.enclosingTube=NULL
actions delete(wire, X)

define trigger cancelTube for class wire
event delete
condition tube(Y), Y.wires={}
actions delete(tube, Y)

- GT: contiene un ciclo (las reglas se accionan unas a otras)
- La regla *CancelWire* borra un alambre (*wire*) en el caso de que ningún tubo lo incluya: así no se puede producir un tubo sin hilos o alambres. Igual sucede con la regla *CancelTube* y así las dos reglas no puede activarse una a otra.
- Algunas simplificaciones se pueden realizar automáticamente por una herramienta; probar el fin requiere la intervención del diseñador

4.4.2 Técnicas de modularización para el diseño activo de la regla

El problema principal con reglas: es no entender su interacción. El entender reglas después de su diseño es absolutamente difícil. La adición de una regla a un conjunto de la regla puede conducir a los comportamientos inesperados. La razón principal es la carencia de la modularización de dispositivos para las reglas activas.

a) Estratificación de la regla

Un principio dominante del diseño para proporcionar la modularización y el control de la regla es la estratificación. La estratificación consiste en la repartición de reglas en estratos tal que cada estrato incluya reglas uniformes. Como consecuencia de esto el comportamiento de la regla se puede abstraer por:

El razonamiento “localmente " en cada estrato individual
El razonamiento “globalmente " del comportamiento a través de estratos

b) Enfoque de la estratificación

La estratificación puede ser:

- De funcionamiento: las reglas activas de un estrato tienen un comportamiento (intentan alcanzar un objetivo específico).
- De aseveración: las reglas activas de un estrato establecen la verdad de los predicados dados (llamados estratos de post-condición).
- Estructural: eventos de reglas activas de estratos son globalmente acíclicos.

El primer caso incluye los otros.

c) Comportamiento de estratificación

- Para cada estrato *S*, se introduce una métrica ***ms.***
- ***ms*** mide la distancia del estado actual de la base de datos desde un punto fijo del estrato,
- Los estratos son ordenados por prioridad.
- La ejecución de las reglas comienza en estratos bajos de prioridad que no perturbe la métrica de estratos de una prioridad más alta.

d) Convergencia local, conservación métrica

- La Métrica ***ms*** del estrato ***S*** es finita, La función ***N*** definida para cada estado de la base de datos: ***ms : DB→N .***
- El estrato ***S*** *localmente converge para:*
  - Reglas *de S*, consideradas en aislamiento, de cualquier estado inicial ***D1*** alcance siempre un estado fijo, denotado por ***D2*** .
  - Si cualquier regla ejecuta su parte de la acción, ***(D2≠D1)***, entonces. ***ms(d2) < ms (D1)***
- ***Sj*** conserva las métricas de ***Si:***
  Si, siempre que cualquier regla de ***Sj*** se ejecute y transforma un estado de la base de datos ***D1*** en un estado de la base de datos ***D2*** entonces está es ***mi(D2)<=mi(D1)***

Ejemplo:

Una compañía recibe y procesa órdenes de los clientes; el proceso de órdenes transforma órdenes pendientes en órdenes validas cuando ciertas condiciones en el crédito del cliente se resuelven. El proceso de orden es realizado automáticamente por reglas activas: se validan las órdenes si están dentro del crédito de los clientes, o si se confían en un cliente.

- Las tablas son las siguientes:

order(order-number,client,amount,status)
trusted(client)
credit(client,amount)

- Las reglas activas son:

***T1**: when inserted order*
if trusted(c), credit(c,x), inserted-order(o,c,a,pending)
then update[order:(o,c,a,OK)],update[credit:(c,x-a)]

***T2**: when updated order.status*
if trusted(c), credit(c,x), new-order(o,c,a1,pending), old-order(o,c,a2,_)
then update[order:(o,c,a1,OK)],update[credit: (c,x-a1+a2)]

***T3**: when inserted order*
If inserted-order(o, c,a, pending), credit(c, x), x > (a1-a2)
then update[order:(o, c,a, OK)], update[credit:(c, x-a)]

***T4**: when updated order.status*
If new-order(o,c,a1,pending), old-order(o,c,a2,_), credit(c, x), x > (a1-a2)
then update[order:(o, c,a1, OK)], update[credit:(c, x-a1+a2)]
metric:count(orders)-count(accepted-orders)

- Modificación de las reglas activas:

    - Asumiendo un cambio de la política de la compañía, debido a la recesión. Cuando el crédito de los clientes esta debajo de 50K, se suspenden sus órdenes.

***T5**: when updated credit*
if credit(c,x), x<50K
then update[order:(o,c,a,pending)]
before T1..T4

    - T5 viola la métrica del estrato.
    - Con las inserciones o actualizaciones de ordenes de millares de clientes confiados, la regla T5 y la regla t2 entran en un ciclo.

- Corrección de la modificación
  - Una corrección obvia es suprimir el hecho de que confían en un cliente como parte de la acción de la regla.

    T5: on update credit
    if credit(c,x), x<50K then update[order:(o,c,a,NO)],
    delete(trusted(c)) before T1..T4

  - T5 es un estrato mas alto que T1..T4:
  - S1:{T5}S2:{T1..T4}

    m1:count(trusted-clients)m2:count(orders)-ount(accepted-orders)

  - S2 preserva la métrica de S1.

4.5 Implementación de reglas activas

Las diferencias sintácticas y semánticas de los diferentes sistemas hacen complejo la traducción. Las implementaciones comerciales de disparadores tienen a menudo límites, como la carencia de recursión y de prioridad, que se solucionan para reproducir la semántica de *Chimera*.

Dos traducciones independientes de la regla, adaptables a cualquier sistema:

Meta-triggering utiliza el motor de la regla del sistema para detectar eventos, pero después lo extiende para reflejar la semántica de las reglas de *Chimera*; todas las características de *Chimera* pueden ser reproducidas.

Macro-triggering asocia disparadores de *Chimera* con prioridad en las reglas agregadas al sistema, para hacer cumplir el orden correcto de la ejecución; las

características como la activación diferida, la atomicidad de las líneas de la transacción y la ejecución de la acción de la regla no se pueden obtener con *Macro-triggering*.

5. VENTAJAS Y DESVENTAJAS DE LAS BASES DE DATOS ACTIVAS

Los Sistemas administradores de Base de Datos Activa están, en parte, "dirigidos por los datos" en lugar de estar dirigidos exclusivamente por programas como los actuales. La lógica de control se codifica en reglas almacenadas en la base de datos en lugar de estar en los propios programas.

5.1 Ventajas

Las principales ventajas de las Bases de Datos Activas son las siguientes:

- Simplifican los programas al descargarlos de aquellos controles que, en realidad, forman parte de la semántica de los datos.

- Consiguen una mayor productividad y un menor mantenimiento ya que las reglas se almacenan y, si es necesario, se modifican una sola vez en el diccionario de la base de datos, en lugar de hacerlo en cada programa.

- Reducen el tráfico de red, pues al almacenar parte de los procedimientos en los servidores se limita la cantidad de información que éstos deben solicitar y/o devolver.

- Facilitan el acceso a la base de datos por los usuarios finales, al almacenar las reglas de actualización en el propio Sistema administrador de Base de Datos Activa. Este podrá preservar la integridad de los datos independientemente de cuál sea el método de acceso empleado, lo que permite a los usuarios finales acceder sin peligro de dañar la base de datos.

5.2 Desventajas

- La tiene una desventaja en la ejecución porque verifica condiciones de reglas que son considerablemente lentas cuando se hace un gran número de comparación entre los elementos. Por consiguiente, alguna clase de mecanismo para acelerar el proceso de verificación de la regla está en gran demanda por mejorar la ejecución global de una. Algunas Bases de Datos Activas hacen uso de discriminación de redes para mejorar la velocidad de verificación de la condición de la regla, pero todavía no es suficiente.

- Al implementar reglas de producción se especifica las reglas en la aplicación usando el lenguaje de programación del DBMS. Este acercamiento crea un sistema más activo, pero los problemas todavía existen. Un problema es que no hay ninguna herramienta regularizada para los lenguajes de programación de DBMS para crear estas reglas. Así, se crean reglas usando cualquier metodología según los deseos de los programadores individuales. No se estandarizan reglas y, por consiguiente, a menudo no puede reutilizarse. El no reutilizar la regla es desventajoso por dos razones. Primero, la creación de la regla consume tiempo y el proceso puede ser tedioso porque deben desarrollarse estructuras de la regla cada vez que una regla se crea. La segunda desventaja relaciona la exactitud. Creando una regla y determinando que está funcionando correctamente están consumiendo tiempo y a veces provoca errores.

6. DISPARADORES

6.1 Historia de los disparadores en las bases de datos

Programadores y usuarios siempre reconocieron el valor de guardar información para su posterior revisión y análisis, pero los sistemas computacionales no ofrecían un soporte adecuado. La demanda constante por parte de los consumidores, sumada al progreso de los proveedores en tecnología de almacenamiento, hicieron que el *software* haya experimentado un crecimiento asombroso en los últimos tiempos y que muestre claros signos de consolidación.

Las bases de datos relacionales separaron los programas de las estructuras de datos subyacentes, con lo cual se pudo acceder a éstas sin necesidad de acudir a los primeros. Un importante efecto colateral de esta ventaja fue el surgimiento resultante de las arquitecturas tipo cliente-servidor, las que ofrecían un control central al igual que la administración de información.

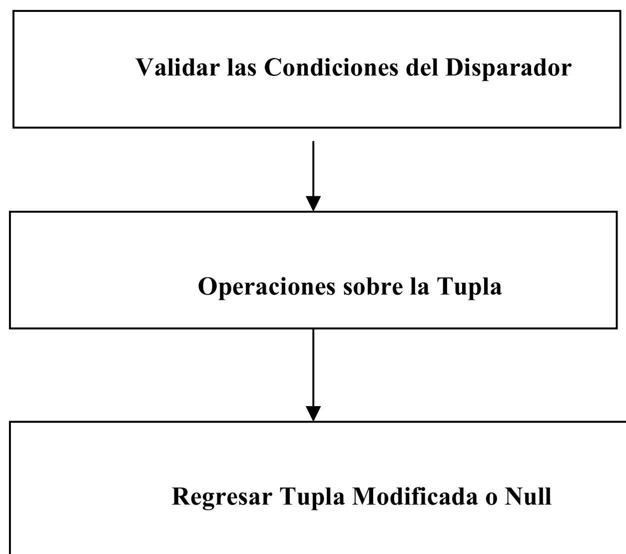
Asimismo, un esquema de información bien diseñado y personalizado requiere de menos espacio de almacenamiento y reduce al mínimo las inconsistencias asociadas con los datos redundantes. Tal vez más importante que las ventajas inherentes al modelo relacional sean las asociadas con el nuevo "super estándar" de la industria: El lenguaje de consulta estructurado *SQL* (*Structured Query Language*), que es utilizado por los programas de aplicaciones para realizar consultas a las bases de datos y ha sido adoptado por los principales proveedores y programadores de aplicaciones de clientes. Un código comprendido por todos brinda un marco mental común para el control, la definición y la manipulación de datos.

A medida que el modelo relacional fue adquiriendo aceptación, aparecieron los grandes proveedores de bases de datos que ampliaron la funcionalidad y mejoraron el desempeño en la lucha por obtener una mayor participación en el mercado. Entre los ejemplos de funciones adicionadas, cabe mencionar los "procedimientos almacenados" (*Store Procedure*), que son colecciones de sentencias *SQL* compiladas y almacenadas en la base de datos. Estos son más rápidos que las sentencias *SQL* normales y reducen el tráfico en la red. También, se destacan los **"disparadores"** (*Triggers*), sentencias *SQL* invocadas automáticamente por el servidor, usados para preservar la integridad de los datos y realizar tareas en relación con cambios de datos, como por ejemplo auditorías o rastreos.

6.2 Facilidad en la especificación de eventos

Los disparadores son los procedimientos especializados en manejo de eventos (*event-driven*) que se almacenan dentro del RDBMS. Cada disparador se asocia a una tabla. Los disparadores se pueden pensar como una forma avanzada de "regla" o de "restricción" escritos usando una forma extendida de *SQL*.

Figura 10. Diagrama de flujo de un disparador



En la figura 10 se puede observar las tres fases que debe llevar un *trigger*. Después de haber validado que el *trigger* se disparó en las condiciones deseadas, se procede ha realizar operaciones sobre la tupla sobre la cual se ejecuta el *trigger*. La tupla puede ser leída, modificada o ser rechazada.

Un disparador no puede ser llamado o ser ejecutado directamente; es ejecutado (o " disparado automáticamente ") por el RDBMS como resultado de una acción, una modificación de los datos de la tabla a la cual se encuentra asociado. Una vez que se ha creado un disparador se ejecuta siempre y cuando ocurra un evento (*update, insert, delete*). Por lo tanto los disparadores son automáticos e implícitos.

Los disparadores son útiles para poner el código en ejecución que se debe ejecutar sobre una base regular debido a un evento predefinido. Utilizando disparadores, el programar y los problemas de integridad de los datos pueden ser eliminados porque el disparador será activado siempre que ocurra el evento. No se necesita recordar programar o planificar una actividad para implementar la lógica en el disparador. Sucede automáticamente en virtud de lo que está en el disparador.

6.3 Ejemplos

Se propone informatizar una biblioteca creando una base de datos que mantenga información de los volúmenes existentes, de los socios inscritos y de los préstamos efectuados. Evidentemente, el objetivo es simplemente ilustrar el sentido y necesidad de *triggers* por lo que el diseño se ha simplificado al máximo evitando cualquier tipo de consideración sobre el modelo de datos. El primer paso será definir las tablas necesarias. Del más sencillo análisis resulta que es necesario, al menos, definir una tabla libros, una segunda autores, una tercera socios, y una cuarta prestamos, que servirá para almacenar cada acción de préstamo que se lleve a cabo. Los disparadores se implementaran en *Sql Server* (Este DBMS es muy popular en el manejo de *triggers*).

Tabla libros:

CREATE TABLE libros
(registro smallint
signatura char(20),titulo varchar(50),
nombreautor varchar(50)*editorial* varchar(20),
fechaentrada datetime)

Tabla *autores*:

CREATE TABLE autores
(idautor smallint
nombreautor varchar(50),
nacionalidad varchar(20))

Tabla *socios*:

CREATE TABLE socios
(idsocio smallint
nombresocio varchar(50),
edad smallint,
direccion varchar(20),
teléfono varchar(10))

Tabla *prestamos*:

CREATE TABLE prestamos
(idprestamo smallintidautor smallint,
idsocio smallint,fechaprestamo datetime)

Existen múltiples situaciones en las que en este ejemplo sería interesante definir un *trigger*. Una primera muestra sería la del control de préstamo. Imagínese que se desea evitar que cada socio pueda tener más de un libro simultáneamente en su poder. Cada vez que se realiza un préstamo nuevo la aplicación cliente solicitará la inserción de una nueva fila en la tabla de préstamos. Se podría escribir un *trigger* que tras cada inserción en esta tabla, buscase si existe alguna otra fila en la misma que corresponda a un préstamo al mismo socio, y que eliminase el nuevo préstamo si ya existía uno aún no terminado. La aplicación cliente podría recibir información en la que examinase si la inserción se había realizado correctamente y presentar, si así se desea un mensaje explicativo. Pero, lo que es más importante, los datos de la base de datos serían coherentes. Vea este ejemplo:

Inserción condicional de un préstamo

```
CREATE TRIGGERprestamocondicional
ON prestamos
FOR INSERT AS
IF
(SELECT COUNT(*) FROM prestamos, inserted
WHERE prestamos.idsocio = inserted. idsocio ) <> @@rowcount
BEGIN
DELETE * FROM prestamos
WHERE prestamos. idprestamo = inserted. idprestamo
END
```

Otra posibilidad es la de gestionar la correspondencia entre la tabla libros y la tabla autores cuando se da de baja un ejemplar por pérdida, sustracción o cualquier otra circunstancia. En este caso el *trigger* deberá examinar, tras la eliminación de un libro, si existe el autor del libro eliminado tiene aún algún volumen en el fondo de la biblioteca, eliminando su entrada en la tabla autores si no es así. La tabla *inserted* almacena una copia de las filas que se han añadido durante la ejecución de una sentencia *insert o* update sobre una tabla que tiene asociado un *trigger.* La tabla *deleted* almacena una copia de las filas eliminadas durante una sentencia *delete* o *update*. Evidentemente, una vez realizada la operación de borrado, las filas desaparecerán de la tabla asociada al *trigger* y sólo aparecerán en la tabla *deleted.* El *trigger* sería similar al siguiente:

Eliminación condicional de un autor

```
CREATE TRIGGER comprobarautor
ON libros
FOR DELETE AS
IF
(SELECT COUNT(*) FROM libros
WHERE libros.idautor = deleted. idautor ) > 0
BEGIN
DELETE * FROM autores
WHERE autores. idautor = deleted. idautor
END
```

CONCLUSIONES

1. Las Bases de Datos Activas son formas inteligentes de base de datos, ellas eliminan la necesidad de muchos de los subsistemas de aplicación que se requieren actualmente para traer bases de datos pasivas hasta estándares razonables de operación. Ejemplos de aplicaciones que se sugieren o son facilitadas por Sistemas de Bases de Datos Activas, pero están más allá del alcance de sistemas de bases de datos pasivos, son sistemas expertos y manejadores de procesos de negocio *(workflow)*.

2. La arquitectura básica de un DBMS activo, esta compuesto por eventos, condiciones, acciones, reglas ECA, y la base de datos.

3. Aunque muchos sistemas pueden utilizar un lenguaje de regla activa similar, la manera en que las reglas se interpretan puede variar extensamente. Pero se tiene que contar con la detección de evento, el método de la ejecución de la regla, la resolución del conflicto, y el modo de acoplamiento.

4. Las reglas activas proporcionan un nuevo método importante para diseñar las bases de datos y el tema está siendo considerado por una gran cantidad de compañías comerciales de las bases de datos.

5. Las reglas son la base de los Sistemas de Administración de Bases de Datos Activas y aparecen bajo muchos nombres: reglas de evento-condición-acción (ECA), disparadores, monitores y alertas.

6. El término integridad en las bases de datos se refiere a asegurar que los datos sean válidos. La especificación de reglas de integridad (llamadas también reglas del negocio) es relativamente compleja, y no era posible definirlas como elementos de la estructura de la base de datos. Era deseable que los DBMS almacenaran las reglas de integridad en el diccionario de datos, de forma que el subsistema de integridad esté constantemente monitoreando las transacciones que realizan los usuarios, asegurando que las reglas se cumplan. Esto es posible actualmente por medio de las bases de datos activas.

7. Actualmente, muchos fabricantes de sistemas administradores de base de datos relacionales empiezan a incorporar algunas funcionalidades propias de Base de Datos Activas, en sus desarrollos, consistentes en la incorporación de disparadores. El principal inconveniente es la falta de estandarización pues cada fabricante decide qué tipos de disparadores va a soportar, cuales no y además, los lenguajes para programarlos son propietarios de cada fabricante. La situación mejorará con la versión de *SQL (SQL3)* pues en él, ya están incluidos los estándares de definición y utilización de disparadores.

8. *Postgres* es un DBMS que es muy completo, ya que maneja un sistema de reglas, y también incorpora disparadores. Cada DBMS presentado tiene su propia forma de funcionar como DBMS activo, *Roke* que es un ambiente para la definición y ejecución de reglas activas construido sobre *Orácle*. *Db2* incorporo disparadores en 1999, en la versión 6. Con esto se puede demostrar que no existe estándar para la definición activa de los DBMS.

9. Por otra parte, existe los DBMS que han surgido llamándose DBMS activos, los cuales se encuentran en este trabajo, como lo es *Chimera, Hipac* es el primer DBMS activo que surgió, *Ode y Polyhedra.*

10. La Comunidad Europea se ha interesado en el estudio de los DBMS activos, y creo un proyecto en el cual se definió la metodología para crear aplicaciones de base de datos, en el cual incorporan la tecnología activa, llamada metodología IDEA.

RECOMENDACIONES

1. Fomentar el deseo de explorar la tecnología de base de datos a su capacidad máxima, en los estudiantes de las carreras de computación y sistemas.

2. Divulgación del conocimiento académico en el desarrollo de aplicaciones basadas en reglas.

3. Usar las características avanzadas de la tecnología de base de datos como objetos y reglas en las aplicaciones.

4. Utilizar Sistemas de Administración de Bases de Dato Activas, en aplicaciones que requieran un control a detalle, de la Seguridad de los datos, ya que ellas ofrecen los elementos necesarios para implementar dicha seguridad.

5. A nivel de aplicaciones cliente/Servidor, la inclusión de las reglas en los Sistemas de Administración de Bases de Datos Activas, permite el descongestionamiento de la red, al realizarse el tratamiento local, esto es una de las razones por las que se deberían aprovechar esta tecnología

BIBLIOGRAFÍA

1. **Active Database Solutions**, Inc - Oracle Consultants. http://www.adsinc.com/welcome.htm.2000.

2. **Active DBMS from FOLDOC.** http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi. Noviembre de 1994.

3. **Active rule**. http://www.ing.unico.it/Idea/presentation/active.htm. agosto 2000.

4. Berndtsson, Mikael .**Ode.** http://www.ida.his.se/ida/adc/bibl/node15.html. Mayo de 1997.

5. Ceri, Stefano. **The active database**. Dipartimento di Elettronica e Informazione. (Italia). http://www.elet.polimi.it/engine/sp.asp. Noviembre de 1998.

6. **Coleman Consulting Inc**. http://www.colemanconsulting.com/~troycci/toc.htm. Mayo de 1997.

7. Cochrane, Pirahesh, Mattos. **Integrating Triggers and Declarative Constraints in SQL Database Systems**. http://citeseer.nj.nec.com/cochrane-integrating.html. 1997.

8. Cho,Heeje. **Active Database Facilities in Ode**. DKE Lab. http://dke.postech.ac.kr/~heeje/seminar/ode/ode.html. Enero 1987.

9. **Databases, Information Systems and Multimedia.** http://www.elet.polimi.it/section/compeng/db/. Noviembre de 1998.

10. Fraternali, Piero. **The Idea Methodology.** http://www.elet.polimi.it/idea/dasfaa/node1.html. Marzo de 1997.

11. **Journal papers**. http://poll.informatik.hu-erlin.de/~dbis/publications/journals.html. Septiembre de 1998.

12. Lockhart, Thomas. **PostgreSQL Programmer's Guide**. http://lucas.hispalinux.es/Postgresql-es/rsantos/navegable/ programmer/programmer.htm. 1996.

13. Meneses, Jairo E. José Abásolo. **ROKE: Un ambiente para la ejecución de reglas Activas**. http://agamenon.uniandes.edu.co/~magister/articulos/inginfo/. Agosto de 1998.

14. Miyazaki Jun, Haruo Yokota. **A Parallel Optimistic Dynamic Optimization of Discrimination Networks for Active Databases**. http://yokota-www.cs.titech.ac.jp/~yokota/abstract/DMIB.html. septiembre 2000.

15. Morgenstern, Matthew. **Active Databases as a Paradigm for Enhanced Computing Environments. USC Information Sciences Institute**. http://www1.acm.org/turing/sigmod/dblp/db/conf/vldb/Morgenstern83.html. Septiembre 2000.

16. Mullins,Craig S. **Triggers and DB2 Version 6**. www.platinum.be/craigm/db2_triggers.htm. Marzo de 1998.

17. Neelakantan, Shashi. **ECA RULES**. www.cise.ufl.edu/~shashin/theses/node6.html. Agosto de 1997.

18. **Ode - Base de datos Activa De Object-Oriented** http://www-db.research.bell-labs.com/project/ode/ode-announce/index.html. Septiembre 2000.

19. **POLYHEDRA**. http://www.polyhedra.com/product.htm. Septiembre. 2000.

20. **Rules – ADBMS**. http://csel.cs.colorado.edu/~barthelm/rules/links.html. 1987.

21. **RULE ECAS**. www.cise.ufl.edu/~shashin/theses/node6.html. Septiembre 2000.

22. **Research areas**. http://www.dbnet.ece.ntua.gr/research/index.html. Marzo de 1997.

23. **The active database central**. www.ida.his.se/ida/adc/. Abril 1997

24. Urban.,Dr. Susan D. **CDOL RULE-BASED LANGUAGE SYNTAX.** http://www.eas.asu.edu/~adood/CDOL_BNF/cdol_bnf.html . Agosto de 1996.

25. Van Den Akker, Johan. **A Design Theory for Active Databases**. http://www.cwi.nl/cwi/projects/adb.html. Marzo de 1996.

26. Westwood, James. **Active database systems**. http://www.almac.co.uk/personal/jwestwood/ActiveDB/index.htm. Septiembre de 2000.