C
265

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

UNA EXTENSION DE TCL/TK AL ALGEBRA RELACIONAL

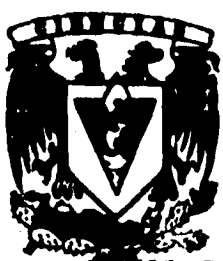
T E S I S

QUE PARA OBTENER EL TITULO DE

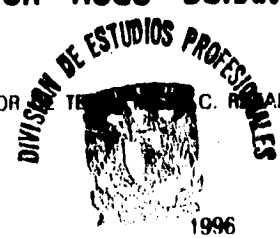
M A T E M A T I C O

P R E S E N T A :

VICTOR HUGO DORANTES GONZALEZ



DIRECTOR DE TESIS: M. en C. RAFAEL MORALES GAMBOA



TESIS CON
FALLA DE ORIGEN

VNIVERMDAD NACIONAL
AVPNMA DE
MEXICO

M. en C. Virginia Abrín Batule
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
P r e s e n t e

Comunicamos a usted que hemos revisado el trabajo de Tesis:

" UNA EXTENSION DE TCL/TK AL ALGEBRA RELACIONAL ".

realizado por VICTOR HUGO DORANTES GONZALEZ

con número de cuenta 8315637-7 , pasante de la carrera de MATEMATICO

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M. EN C. RAFAEL MORALES GAMBOA
Propietario	M. EN C. MARIA GUADALUPE ELENA IBARGUENGOITIA GONZALEZ
Propietario	DRA. HANNA OKTABA
Suplente	M. EN C. GUSTAVO ARTURO MARQUEZ FLORES
Suplente	MAT. ANA LUISA SOLIS GONZALEZ COSIO

M. EN C. ALEJANDRO BRAVO MOJICA
Consejo Departamental de Matemáticas

Agradecimientos

A mi esposa Laura por todo su amor y por lo que hemos compartido, siendo este trabajo uno más de esos momentos importantes.

A mi hija Martha Pamela quien es la luz más resplandeciente en el cielo de mi vida.

A mi familia, mi Mamá, Martha, Carlos, Beto, Adriana, Angel (2), en fin, cada uno porque viven en mi corazón. A mi tía Antonia†, a David†, a Sergio†, porque habrá un tiempo en el cual pueda darle de nuevo un beso a mi tía y un abrazo a mis amigos.

A Dennis, Linus, Jonh, Donald, Richard, Brian, y toda la gente que ha hecho el camino por el cual marcha la computación. A todos y cada uno de mis profesores, muy en especial a Rafael por todo lo que he aprendido de él y por su paciencia para guiar este trabajo. A la Facultad de Ciencias y la UNAM.

Índice General

Lista de Figuras

Introducción

Tcl es uno de los lenguajes de programación e interacción que más ha llamado la atención de la comunidad internacional recientemente, el cual tiene como una de sus características más importantes la posibilidad de extender el conjunto de comandos que posee. El objetivo central de este trabajo consiste en extender el lenguaje Tcl con una selección de comandos de alto nivel basados en la definición formal del álgebra relacional. Tal extensión permite ampliar las capacidades de Tcl y orientarlo hacia el desarrollo de sistemas que requieren del manejo de bases de datos.

Las reglas sintácticas de Tcl y de los nuevos operadores permite su composición, permitiendo de nuevo un estricto apego a la parte formal. Se utiliza el álgebra relacional basada en los ocho operadores básicos, mas aquellos operadores que dan completez al álgebra misma y otros comandos que facilitan el desarrollo de aplicaciones.

El poder ofrecer un superconjunto de operadores del álgebra relacional, permite trabajar con independencia del manejador de bases de datos en particular y su correspondiente lenguaje de definición y manejo de datos. Los nuevos operadores se encargan de interactuar con los manejadores de bases de datos que se encuentran en el nivel más bajo del esquema de trabajo; transforman la forma sintáctica del operador en Tcl a un predicado de consulta en el lenguaje del manejador. Para mostrar que el esquema propuesto es independiente del manejador de bases de datos, se utilizan dos manejadores distintos que tiene orígenes, objetivos y lenguajes de definición y manipulación de datos totalmente diferentes: **INGRES** y **mSQL**.

Para efectos de mostrar el uso de la extensión de Tcl y las capacidades de **Tk** (la extensión de Tcl para el desarrollo de interfaces gráficas), se desarrolla una interfaz gráfica para el manejo de bases de datos. Este programa puede ser considerado una interfaz al álgebra relacional, con un enfoque didáctico; *i.e.* tiene la finalidad de ser útil como una herramienta para la enseñanza de esta parte fundamental del modelo relacional.

El primer capítulo contiene una descripción del modelo relacional y del álgebra relacional.

El segundo capítulo muestra a los manejadores de bases de datos desde sus orígenes, esquemas de trabajo y sobre todo sus lenguajes de definición y manipulación de datos.

El tercero muestra los componentes principales del lenguaje de comandos Tcl y de su extensión para el desarrollo de interfaces gráficas (*Toolkit*) Tk.

El cuarto capítulo contiene el trabajo desarrollado para implementar la extensión de Tcl al álgebra relacional llamada **Alg**, mientras que el quinto capítulo contiene la interfaz *Alg-Tk* que se desarrolló basada en Alg y Tk.

Capítulo 1
Álgebra relacional

En este capítulo se presentan los fundamentos del modelo relacional, para más adelante mostrar los manejadores de bases de datos que se fundamentan en tal modelo. La idea es definir lo que es un modelo de datos, presentar el modelo relacional en términos de la definición anterior y posteriormente analizar a detalle los conceptos y definiciones del álgebra relacional.

1.1 Modelo de datos

"Un modelo de datos es un sistema formal y abstracto que permite describir los datos de acuerdo con reglas y convenios predefinidos. Es formal pues los objetos del sistema se manipulan siguiendo reglas perfectamente definidas y utilizando exclusivamente los operadores definidos en el sistema, independientemente de lo que estos objetos y operadores puedan significar."[RIVERO88]

Según Cood, en [COOD81]:

"Un modelo de datos es una combinación de tres componentes:

1) una colección de estructuras de datos (los bloques constructores de cualquier base de datos que conforman el modelo);

2) una colección de operadores o reglas de inferencia, los cuales pueden ser aplicados a cualquier instancia de los tipos de datos listados en (1), para consultar o derivar datos de cualquier parte de estas estructuras en cualquier combinación deseada;

3) una colección de reglas generales de integridad, las cuales explícita o implícitamente definen un conjunto de estados consistentes — estas reglas algunas veces son expresadas como reglas de insertar-actualizar-borrar."

Un modelo de datos puede ser usado de las siguientes maneras:

i) como una herramienta para especificar los tipos de datos y la organización de los mismos que son permisibles en una base de datos específica;

ii) como una base para el desarrollo de una metodología general de diseño para las bases de datos;

iii) como una base para el desarrollo de familias de lenguajes de alto nivel para manipulación de consultas (*querys*) y datos;

iv) como el elemento clave en el diseño de la arquitectura de un manejador de bases de datos.

El primer modelo de datos desarrollado con toda la formalidad que esto implica fue el *modelo relacional*, en 1969, mucho antes incluso que los modelos jerárquicos y de red. A pesar de que los sistemas jerárquicos y de red como software para manejar bases de datos son previos al modelo relacional, no fue sino hasta 1973 que los modelos de tales sistemas fueron definidos, apenas unos cuantos años antes de que estos sistemas empezaran a caer en desuso.

1.2 El modelo relacional

Con base en las definiciones anterior sobre lo que es un modelo de datos, es posible analizar el modelo relacional en términos de los tres componentes ya mencionados: estructuras de datos, operadores y reglas de integridad.

1.2.1 Estructuras de datos

Las estructuras de datos que se manejan en el modelo relacional corresponden a los conceptos de relación, entidad, atributo y dominio, los cuales de introducen aquí intencionalmente:

Relación. Por una relación se entiende una colección o grupo de objetos que tienen en común un conjunto de características o atributos.

Entidad. Es una unidad de datos en una relación con un conjunto finito de atributos. Es también conocido como *n-ada*, a raíz de que consiste de *n-valores*, uno por cada atributo.

Atributo. También llamado *característica*, cada atributo de una relación tiene asociado un dominio en el cual toma sus valores.

Dominio. Es un conjunto de valores que puede tomar un atributo en una relación.

La notación más usual para denotar las relaciones en términos de estos conceptos es:

$$Relación = \{atributo_1, atributo_2, \ldots, atributo_n\}$$

por ejemplo

$$Alumnos = \left\{ \begin{array}{cccc} \textbf{No\_Cta}, & \textbf{Nombre}, & \textbf{Ap\_Paterno}, & \textbf{Ap\_Materno} \\ 8315637-7, & Víctor Hugo, & Dorantes, & González \\ 8105765-9, & Magnolia, & Avalos, & Vázquez \end{array} \right\}$$

Y para establecer la conexión entre un *atributo* y un *dominio* podemos usar la siguiente notación:

$$D_{atributo} = \{valor_1, valor_2, \ldots, valor_m\}$$

por ejemplo

$$D_{No\_Cta} = \{8315677-7, 8409695-9, \ldots, 8759452-6\}$$

Una forma de implementar las relaciones en una computadora, es a través de tablas de valores, de forma que se tienen las siguientes equivalencias: un atributo corresponde al encabezado de una columna, una *n-ada* equivale a un renglón de la tabla y por supuesto una relación equivale a la tabla misma. En capítulos posteriores, suponiendo natural esta equivalencia de términos, se utilizarán de forma indistinta.

1.2.2 Reglas de integridad

Los conceptos básicos de integridad en el modelo relacional son el de ***llave primaria***, ***llave foránea***, ***valores nulos*** y un par de ***reglas de integridad***.

Una ***llave primaria*** es uno o un conjunto de atributos que permiten identificar a las *n-adas* de manera única en cualquier momento.

Una ***llave foránea*** de una relación es un atributo que hace ***referencia*** a una ***llave primaria*** de otra relación; esto da pie a que una relación pueda tener varias ***llaves foráneas***.

Un *valor nulo* es un valor que está fuera de la definición de cualquier dominio el cual permite dejar el valor del atributo "*latente*", su uso es frecuente en las siguientes situaciones:

i) Cuando se crea una *n-ada* y no se conocen todos los valores de cada uno de los atributos.

ii) Cuando se agrega un atributo a una relación ya existente.

iii) Para no tomarse en cuenta al hacer cálculos numéricos.

Las dos *reglas de integridad* tienen que ver precisamente con los conceptos antes mencionados y son:

Integridad de Relaciones. Ningún atributo que forme parte de una *llave primaria* puede aceptar *valores nulos.*

Integridad Referencial. Al tener una relación Q con *llave primaria* A de dominio D y otra relación R con atributo A que no es llave primaria de R, entonces cualquier valor en el atributo A en R debe ser:

i) nulo, o

ii) un valor que esté en el atributo A de la llave primaria de una *n-ada* en la relación Q

1.2.3 Operadores

Los operadores del modelo relacional son de dos tipos: *operadores de actualización* y los *operadores del álgebra relacional.* Los operadores del **álgebra relacional** serán analizados en la sección 1.3.3, mientras tanto mostraremos los de **actualización**[†].

Las operaciones válidas para actualización de los valores de las ***n-adas*** son las de *borrar*, *agregar* o *modificar*. El manejo de las ***llaves primarias*** y ***foráneas*** incide directamente en procurar que no se violen las reglas de integridad, al determinar cómo han de manejarse los operadores de manera que al aplicar cualquiera de estas operaciones no se produzcan inconsistencias.

[†] *Los operadores de actualización se basan en dos suposiciones: a) las relaciones se hayan contenidas en una "base de datos", dentro de la cual son modificadas, b) al cambiar un elemento de la relación sigue siendo "la misma", pero "cambiada"; es decir, pasó de un estado a otro, pero sigue siendo la misma relación.*

Las reglas son las siguientes:

Reglas para agregar. Al insertar una *n-ada* en una relación, el valor de un atributo que sea *llave foránea* puede ser *nulo*, o algún valor del atributo de la *llave primaria* en la relación correspondiente.

Reglas para borrar. Si se tiene una *n-ada* en una relación R_1 con un atributo A_i como *llave primaria*, y otra relación R_2 que tiene ese mismo atributo A_i pero como *llave foránea*, tenemos 3 casos:

i) Borrado restringido. No se puede borrar la *n-ada* en la relación R_1 cuya *llave primaria* tenga un valor que en la relación R_2 exista como uno de los valores de la *llave foránea*.

ii) Borrado en cascada. Al borrar una *n-ada* en la relación R_1 con cierto valor en la *llave primaria*, se borrarán todas las *n-adas* en R_2 que tengan ese mismo valor en la *llave foránea*.

iii) Borrado por nulificación. Al borrar una *n-ada* en la relación R_1, a todas las *n-adas* con el mismo valor en la relación R_2 se les asigna un *valor nulo* en el atributo de la *llave foránea*.

Reglas para modificar. Tenemos dos opciones:

i) Modificación en cascada. Al modificar una *llave primaria* en R_1 se le cambian los valores correspondientes en la *llave foránea* de R_2.

ii) Modificación por nulificación. Al cambiar los valores de la *llave primaria* en R_1 a los correspondientes valores en la llave foránea de R_2 se les pone un *valor nulo*.

Los esquemas de nulificación, en cascada y restringido tienen una aplicación lógica en cuanto a cual de ellas utilizar, esto es, quien decide el esquema a utilizar es quien genera las relaciones y quien sabe cuales son las dependencias entre una relación o atributo con otros, ademas podemos pensar que por periodos o situaciones particulares podemos cambiar de uno a otro esquema.

Ademas la implementación del modelo relacional en un manejador de bases de datos no obedece al 100% con todo el modelo y en particular necesita uno ubicar cual de estos esquemas permite (si es que tiene alguno).

1.3 Álgebra relacional

1.3.1 Relaciones.

Una *relación* R es una colección de dos partes, $R = (H, T)$ con un *encabezado* (*Header*) H y un *cuerpo* T. El *encabezado* consiste de un conjunto de parejas ordenadas†

$$H = \{(A_1 : D_1), (A_2 : D_2), \ldots, (A_n : D_n)\}$$

de atributos y dominios tal que para cada atributo A_j se tiene una asociación con su correspondiente dominio D_j, con $j = 1, 2, \ldots, n$. Los atributos deben ser todos distintos; es decir, $A_i \neq A_j$ si $i \neq j$.

El *cuerpo* consiste de un conjunto de *n-adas*, donde cada *n-ada* se ve como un conjunto de parejas (*atributo* : *valor*)‡

$$T_i = \{(A_{i1} : V_{i1}), (A_{i2} : V_{i2}), \ldots, (A_{in} : V_{in})\}$$

(i=1, 2, ..., m) donde m es el número de *n-adas* del conjunto, $A_{ij} = A_j$ y $V_{ij} \in D_j$; *i.e.* cada valor que toma un atributo A_j está contenido en el correspondiente dominio D_j.

A los valores m y n se les conoce como la ***cardinalidad*** y el ***grado*** de la relación respectivamente, la cardinalidad es de esperarse que varíe con el paso del tiempo, mientras que el grado no; en este sentido hay que remarcar que en contexto de una base de datos que contiene un conjunto de relaciones estos conceptos de cardinalidad y grado se aplican a cada estado de la relación con respecto a un cierto tiempo t. En estos términos, una relación de grado uno se le conoce como ***unaria***, a una de grado dos se le conoce como ***binaria***, etcétera.

Es de resaltar el hecho de que estas definiciones se hacen en base a los conceptos matemáticos de la teoría de conjuntos y que las operaciones mismas de los conjuntos forman parte del álgebra relacional como se verá a continuación.

---

† *Se usan estos índices los cuales solo marcan una relación entre un atributo y su correspondiente dominio, pero no hay ninguna relación de orden entre las parejas.*

‡ *De nuevo es importante resaltar que solo existe una relación entre los valores que toma un atributo y el atributo mismo, pero ninguna relación de orden entre estos.*

1.3.2 Propiedades de las relaciones

De la definición de relación se infieren las siguientes propiedades:

i) No hay n-adas duplicadas
Esta propiedad se obtiene del hecho de que el cuerpo de una relación es un conjunto (en el sentido matemático) y los conjuntos por definición no contienen elementos repetidos. De aquí podemos desprender un corolario importante: el conjunto total de atributos de la relación es siempre una *llave primaria.*

ii) No hay orden en las n-adas
De nuevo esta propiedad se desprende al observar que el cuerpo de la relación es un conjunto no ordenado; esto quiere decir que a ninguna *n-ada* se le puede asignar el título o nombre de la *primera n-ada*, segunda, o el número que sea de la relación y por supuesto tampoco existe el concepto de la *n-ada siguiente* en la relación misma. †

iii) No hay orden en los atributos
Esta propiedad se sigue del hecho de que el encabezado de la relación está definido como un conjunto. Como es de esperarse, no existen los conceptos del *primer* ó *n-ésimo* atributo, ni del *siguiente* atributo. De manera analoga se puede manejar una relación que *controle* tal orden.

iv) Todos los valores de los atributos son atómicos
Esta propiedad debería decir: *todos los valores de los atributos sencillos son atómicos.* Esto es una consecuencia al hecho de que los dominios son sencillos – *i.e.* contienen valores atómicos solamente (si se tuviesen atributos compuestos, éstos siempre pueden verse como la concatenación de atributos sencillos).

Esta última propiedad implica que todas las relaciones están *normalizadas*‡ en lo que al *modelo relacional* concierne, de aquí que en el modelo relacional al hablar de una relación siempre se tiene en mente que es una relación normalizada.

---

† *Es importante señalar que el hecho de que se usen indices para enumerar los atributos o n-adas no preestablece un orden implicito entre estas, sino que es mera notación.*

‡ *El término "normalizadas" se refiere en este caso a la primera forma normal que definió Cood. Las formas normales son condiciones que se establecen sobre las relaciones, las cuales al diseñar un sistema y definir la estructura lógica de las relaciones se emplean para quitar de ellas problemas de redundancia y establecer de forma clara las dependencias funcionales entre los atributos en las relaciones. La definición original de Codd [Cood72] solo incluye tres formas normales las cuales pueden consultarse a detalle (junto con otras) en el capítulo 21 del [Date91].*

1.3.3 Operadores del álgebra relacional

Para los ejemplos mostrados en los operadores utilizaremos las siguientes relaciones:

r:

A	*B*	*C*
a1	*b1*	*c1*
a1	*b2*	*c1*
a2	*b1*	*c2*

s:

A	*B*	*C*
a1	*b1*	*c1*
a2	*b2*	*c1*
a2	*b2*	*c2*

q:

D	*E*	*F*
d1	*e1*	*f1*
d2	*e2*	*f1*
d2	*e2*	*f2*

p:

D	*E*	*F*	*G*	*H*
d1	*e1*	*f1*	*g1*	*h2*
d2	*e2*	*f1*	*g2*	*h1*
d2	*e2*	*f2*	*g1*	*h1*

- **Unión**
  Construye una relación consistente en todas las *n-adas* que forman parte de cada una de las dos relaciones especificadas.

  En términos abstractos podemos decir que este operador es una función que toma como argumentos un par de relaciones y da como resultado otra relación:

$$Unión: R \times R \longmapsto R$$

$$Unión(\langle H, R\rangle, \langle H, S\rangle) \longmapsto \langle H, R \cup S\rangle$$

  Las relaciones R y S deben tener el mismo encabezado, al igual que la relación resultado.

  Por ejemplo:

Unión(*r*, *s*):

A	*B*	*C*
a1	*b1*	*c1*
a1	*b2*	*c1*
a2	*b1*	*c2*
a2	*b2*	*c1*
a2	*b2*	*c2*

- **Intersección**
  Construye una relación consistente en aquellas *n-adas* que aparecen tanto en la primera como en la segunda relación dada.

$$Inter: R \times R \longmapsto R$$

$$Inter(\langle H, R\rangle, \langle H, S\rangle) \longmapsto \langle H, R \cap S\rangle$$

  Observamos que tanto las relaciones R y S como la relación resultante tienen el mismo encabezado.

Por ejemplo:

$Inter(r, s)$:

A	B	C
$a1$	$b1$	$c1$

- **Diferencia**
  Se construye una relación consistente en aquellas *n-adas* que pertenecen a la primera relación pero no a la segunda relación especificada.

$$Dif: R \times R \longmapsto R$$

$$Dif(\langle H, R\rangle, \langle H, S\rangle) \longmapsto \langle H, R - S\rangle$$

De nuevo, el encabezado de las relaciones R y S, al igual que la relación resultante, tienen el mismo encabezado.

Por ejemplo:

$Dif(r, s)$:

A	B	C
$a1$	$b2$	$c1$
$a2$	$b1$	$c2$

- **Producto Cartesiano**
  Construye una relación consistente en todas las posibles combinaciones de *n-adas*.

  Este operador es una función que recibe una relación R con encabezado $\{(A_1 : D_1), (A_2 : D_2), \ldots, (A_n : D_n)\}$ y otra relación S con encabezado $\{(B_1 : D_1), (B_2 : D_2), \ldots, (B_m : D_m)\}$ para generar una relación que tenga encabezado $\{(A_1{:}D_1), (A_2{:}D_2), \ldots, (A_n{:}D_n), (B_1{:}D_1), (B_2{:}D_2), \ldots, (B_m{:}D_m)\}$[†]; i.e. la concatenación de los encabezados de R y S.

$$Cruz: R \times R \longmapsto R$$

$$Cruz(\langle H, R\rangle, \langle H', S\rangle) \longmapsto \langle (H, H'), R'\rangle$$

[†] *Es necesario que $A_i \neq B_j$, aunque siempre se puede renombrar los atributos idénticos*

Por ejemplo:

$Cruz(r, q)$:

A	B	C	D	E	F
$a1$	$b1$	$c1$	$d1$	$e1$	$f1$
$a1$	$b1$	$c1$	$d2$	$e2$	$f1$
$a1$	$b1$	$c1$	$d2$	$e2$	$f2$
$a1$	$b2$	$c1$	$d1$	$e1$	$f1$
$a1$	$b2$	$c1$	$d2$	$e2$	$f1$
$a1$	$b2$	$c1$	$d2$	$e2$	$f2$
$a2$	$b1$	$c2$	$d1$	$e1$	$f1$
$a2$	$b1$	$c2$	$d2$	$e2$	$f1$
$a2$	$b1$	$c2$	$d2$	$e2$	$f2$

- **Restricción**

  Extrae aquellas *n-adas* de una relación que satisfacen una condición especifica.

  Este operador es una función que toma una relación y una condición y devuelve una relación con todas aquellas *n-adas* que hayan cumplido con la condición.†

$$Rest\colon R \times Condición \longmapsto R$$

$$Rest(\langle H, R\rangle, Condición) \longmapsto (H, R_{(Cond)})$$

Por ejemplo:

$Rest\langle r, B = b_1\rangle$:

A	B	C
$a1$	$b1$	$c1$
$a2$	$b1$	$c2$

- **Proyección**

  Extrae aquellos atributos especificados de una relación dada.

  Formalmente es una función que toma una relación que tiene un encabezado de n atributos y un conjunto de m atributos (que están dentro del encabezado de la relación dada), generando una relación con un encabezado con los m atributos dados.

$$Proy\colon R \times (atrib_1, atrib2, \ldots, atrib_m) \longmapsto R$$

$$Proy(\langle H, R\rangle, \{atrib_1, atrib2, \ldots, atrib_m\}) \longmapsto \langle H', R_{(atribs)}\rangle$$

† *Por condición entenderemos una expresión formada por operadores lógicos y de comparación entre los atributos y valores dentro los dominios correspondientes.*

Por ejemplo:

$Proy(r, \{B, C\})$:

B	C
$b1$	$c1$
$b2$	$c1$
$b1$	$c2$

- **Theta-Unión. (Join)**
  Construye una relación a partir de dos relaciones las cuales tienen conjuntos ajenos de atributos, a los cuales se les establece un condición lógica que permite unir las relaciones a través de estos atributos.

  Este operador es una función que toma dos relaciones y un conjunto de condiciones de comparación entre atributos de una y otra relación, tal condición sirve para establecer una conexión lógica entre las relaciones (de manera natural se supone que los atributos correspondientes estan sumergidos en los mismos dominios); de manera que la relación generada es una combinación de las *n-adas* de ambas relaciones pero que en los atributos que estan involucrados en la condición, cumplen con ésta.

$$\theta Unión: R_1 \times R_2 \times (Condición) \longmapsto R$$

$$\theta Unión(\langle H, R\rangle, \langle H', S\rangle, \{Condición\}): \longmapsto \langle H'', R'S\rangle$$

Por ejemplo, si renombramos como D, E y F los atributos A, B, C, de la relación s para cumplir con la condición de este operador. Es de observarse que si la relación R tiene n atributos y la relación S tiene m atributos, entonces la relación resultante tiene $n + m - j$ atributos, donde j es el número de atributos diferentes que estan involucrados en la condición.

Ejemplo:

$\theta Unión(r, s, \{A = D\})$:

A	B	C	E	F
$a1$	$b1$	$c1$	$b1$	$c1$
$a1$	$b2$	$c1$	$b1$	$c1$
$a2$	$b1$	$c2$	$b2$	$c1$
$a2$	$b1$	$c2$	$b2$	$c2$

- **División**
  Este operador toma dos relaciones y construye una relación consistente de todos los atributos de la primera relación que no estan en la segunda relación.

  Este operador es una función que toma dos relaciones, una R con encabezado $\{(A_1 : D_1), (A_2 : D_2), \ldots, (A_n : D_n), (B_{n+1} : D_{n+1}), \ldots, (B_m : D_m)\}$ y una S con encabezado $\{(A_1 : D_1), (A_2 : D_2), \ldots, (A_n : D_n)\}$ y regresa una relación de grado $m - n$ con encabezado $\{(B_{n+1} : D_{n+1}), \ldots, (B_m : D_m)\}$.

$$Div: R \times R \longmapsto R$$

$$Div(\langle (HX, HY), R \rangle, \langle HX, S \rangle) \longmapsto \langle HY, R/S \rangle$$

  Por ejemplo:

  Div(p, q):

G	H
g1	*h2*
g2	*h1*
g1	*h1*

Como se puede apreciar, el resultado de cada operador es una relación. Esta es la llamada ***propiedad de la cerradura*** y resulta muy importante porque el resultado de un operador puede ser un argumento para otro operador; *i.e.* permite la composición de los operadores.

Capítulo 2

Manejadores de bases de datos

Sin duda que una de las principales aplicaciones de la computación son las *bases de datos*, las cuales se encuentran en un gran número de lugares como son bancos, comercios, escuelas, etcétera. El tipo de aplicaciones para las cuales se usan son muy importantes ya que normalmente la información que manejan representan los ingresos o egresos de alguna compañia, o bien las calificaciones de una escuela, o el control de inventarios.

Cuando se empezaron a utilizar los recursos de cómputo para realizar sistemas administrativos, lo usual era desarrollar el sistema basado en archivos, lo cual presentaba serios problemas de compatibilidad, implementación, mantenimiento y crecimiento; eran poco confiables y demasiado grandes. Los conceptos de bases de datos empezaron a cobrar importancia por los beneficios que se podían obtener éstos son entre otros:

i) Poder compartir la información (la base de datos) entre todo un conjunto de aplicaciones.

ii) Ser consistentes y evitar redundancia en la información.

iii) Poder delegar y centralizar ciertas responsabilidades como la administración de la información y su seguridad.

Esto planteó un modelo de trabajo en el cual podemos ubicar dos componentes primarios como son el **hardware**, el cual por supuesto consiste del equipo de cómputo con que se cuenta (CPU, memoria, etcétera) y que es donde se guarda y maneja físicamente la información; y el segundo componente es a nivel **software**, el cual consiste básicamente de un *sistema manejador de bases de datos* (*SMBD*) el cual se encarga de manejar y administrar los datos en la máquina.

Se distinguen tres niveles de usuarios de un manejador de bases de datos: *administrador de la base de datos*, los *programadores de aplicaciones* y los *usuarios finales*. Rápidamente podemos decir que el *usuario final* básicamente tiene funciones de *consulta*, *actualización* y *creación* de datos. El programador de aplicaciones desarrolla los programas que utilizan los usuarios finales para realizar sus operaciones sobre la base

de datos. Y finalmente el administrador de la base de datos controla las estructuras de las bases de datos, sus respaldos e integridad y los mecanismos de acceso.

Los *SMBD* definen dentro de su arquitectura tanto un lenguaje de definición de datos (*LDD*) como un lenguaje de manipulación de datos (*LMD*) que en la implementación normalmente están fusionados en un solo lenguaje. El *LDD* sirve básicamente para definir las estructuras de las bases de datos, mientras que el *LMD* sirve como el lenguaje de acceso y manejo de las bases de datos.

Normalmente también los *SMBD* proporcionan al menos dos maneras de interacción con ellos: de forma interactiva y a través de programación *externa*. Es decir, uno puede trabajar con el *SMBD* comando por comando o bien por medio de todo un programa que permita una interacción más elaborada.

Resumiendo, la característica que ha llevado a desarrollar sistemas basados en las bases de datos es la facilidad de abstraer el manejo de la información para evitarse problemas de consistencia, integridad y redundancia; de esta forma las tareas de seguridad, inconsistencia e integridad de la información se manejan de forma independiente a la aplicación deseada.

Ahora bien todas estas virtudes que proponen las bases de datos son posibles gracias a que los *SMBD* fundamentan su funcionamiento en un modelo de datos como el mencionado en el capítulo anterior; esto quiere decir que las *bases de datos* han tenido el éxito actual debido a los *fundamentos matemáticos* en los que se sustentan[IBAR94].

A más de veinte años de la definición del *modelo de datos relacional* [CODD70] éste ha llegado a ser el modelo de datos para más del noventa por ciento del mercado de los *manejadores de bases de datos*. El presente trabajo de tesis se desarrolló utilizando como *manejadores de bases de datos* a **Ingres** (versión universitaria de dominio público), un manejador clásico por sus orígenes y objetivos con un *LMD* llamado **QUEL** (el cual casi no se usa en la versión comercial); y **mSQL** (que también es de dominio publico), un manejador que tiene el requisito principal de los manejadores comerciales: **SQL**, el *LMD* más estandar en el mercado.

En la Facultad de Ciencias no se contaba con manejadores de bases de datos para equipos con sistema operativo *UNIX*, de forma que se recurrió a **Internet** para obtener estos manejadores de dominio público, los cuales han sido compilados e instalados en equipos con diversas arquitecturas.

2.1 Ingres

INGRES (Interactive Graphics and **R**etrieval System) es un *SMBD* basado en el modelo relacional, el cual fue implementado inicialmente en una computadora PDP11 con el sistema operativo *UNIX*. El proyecto donde surgió *INGRES* fue iniciado a finales de 1973 por Michael Stonebraker y Eugene Wong en la Universidad de California en Berkeley.

El código de *INGRES* está escrito en el lenguaje de programación **C** (el mismo con el que se desarrolló *UNIX*), y la parte del análisis sintáctico fue implementada con *YACC* (Yet Another Compiler to Compiler), una herramienta para construir compiladores que forma parte de las utilerías de *UNIX*.

Resulta por demás interesante el desarrollo y evolución de este manejador. La documentación al respecto [STON76], [STON80], [STON81], [STON83], [HAWT79], [ALLMAN76], [ROWE90], [YOUSS79], muestra como a partir de la definición del modelo relacional [CODD70] el proyecto *INGRES* inició desde la recopilación y el análisis de los trabajos realizados hasta el momento en el modelo relacional para después pasar a una etapa de división de trabajo, la cual comprendió la definición del lenguaje de manipulación de datos *LMD*, encabezado por Wong, y por otro lado la evaluación y análisis del sistema operativo y lenguaje de programación a utilizar, que fue evaluado por Stonebraker.

Esta forma de trabajar generó por un lado el lenguaje de manipulación de datos llamado **QUEL** (**Q**uery **L**anguage), el cual tomó ciertas ideas del lenguaje DSL/Alpha descrito en [CODD71]. Mientras que, como ya se mencionó, *UNIX* les resultó atractivo como sistema operativo por el manejo *natural* de multiples procesos y su estructura de archivos de donde tomaron muchas ideas.

La plataforma que significó *UNIX* fue de gran influencia, empezando porque tomaron la idea de un *superusuario* (el administrador del SMBD) junto con un sistema de archivos de tipo jerárquico, además que *UNIX* permitió delegar ciertas tareas importantes como el manejo de la memoria. El proceso principal de *INGRES*, llamado también `ingres`, tuvo que ser dividido en 4 procesos (más adelante 3) debido a las limitaciones en cuanto al tamaño de la memoria, de nuevo aprovechando una de las facilidades principales de *UNIX* que son los *pipes*, el cual permite la comunicación entre procesos. La estructura del proceso `ingres` es como sigue:

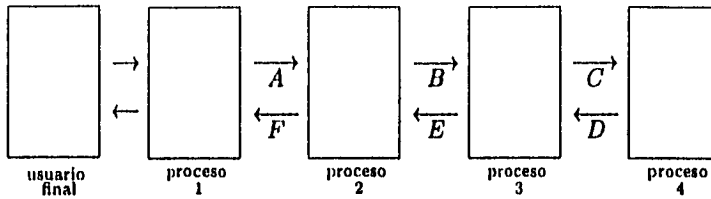


Figura 2.1. Estructura de Procesos de INGRES

En la estructura general podemos observar que de izquierda a derecha (*A*, *B*, *C*) entre los pipes se envian las instrucciones a procesar, mientras de que derecha a izquierda (*F*, *E*, *D*) se manejan los mensajes de error y los resultados de las consultas.

El *proceso 1* es un proceso que se encarga de la interacción con el usuario, *i.e.* que lee cada uno de los comandos y mediante el *pipe A* manda a procesar el comando y por el *pipe F* recibe el resultado del comando y se encarga de presentarlo al usuario.

El *proceso 2* contiene un analizador léxico, un analizador sintáctico, rutinas de control de integridad y un controlador de concurrencia. El proceso le envía los tokens resultado del analizador sintáctico al proceso siguiente a través del *pipe B*.

El *proceso 3* acepta los tokens del comando a ejecutar y manda llamar a las rutinas correspondientes a los comandos *CONSULTA* (*RETRIEVE*), *REEMPLAZA* (*REPLACE*), *BORRA* (*DELETE*), y *AÑADE* (*APPEND*). De hecho este es el proceso principal de la estructura ya que en él se llevan a cabo las pricipales operaciones que tienen que ver con la base de datos y su consulta.

Finalmente, en el *proceso 4* residen un conjunto de comandos de tipo *utilerías* como *CREATE*, *DESTROY*, etcétera. Algunas observaciones finales sobre esta estructura de procesos y el por qué de la misma:

1) Las limitaciones en cuanto al tamaño de la memoria (o espacio de direccionamiento) que se tenían en *UNIX* (64K) al momento del diseño del sistema, junto con la idea de no tener un proceso tan grande como para que no cupiera en la memoria, fueron las justificaciones del por qué cuatro procesos.

2) El flujo del control entre los *pipes* se manejó de una manera muy sencilla; los comandos *fluyen* hacia la derecha y los datos y errores hacia la izquierda. Esto permite una forma de control y sincronización de los procesos bastante aceptable.

3) La estructura permite aislar el proceso 1 (monitor), de manera que este pueda ser sustituido o reemplazado por otro de manera que se puedan generan aplicaciones que no dependan del manejador de bases de datos.

2.1.1 QUEL

El conjunto de comandos de QUEL podemos dividirlo en comandos de *UNIX* para las operaciones a nivel de las bases de datos y comandos dentro de **ingres** que sirven para el control y consulta de las relaciones. QUEL utiliza un mecanismo de asignar una relación a una variable a través de la instrucción **range of variable is relación**.

CREATEDB

createdb nombre_bd

Este comando se puede ejecutar a nivel *UNIX* o como parte de EQUEL; crea una base de datos generando todo un conjunto de relaciones orientadas a administrar las relaciones que se definan posteriormente.

ejemplo:

```
loltun:$ creatdb s-escolares
modifying relation
modifying attribute
modifying indexes
modifying tree
modifying protect
modifying integrities
modifying rdelim
sysmod done
loltun:$
```

DESTROYDB

destroydb nombre_bd

Este comando se puede ejecutar a nivel *UNIX* o como parte de EQUEL; este se encarga de eliminar las relaciones y la base de datos misma.

ejemplo:

```
loltun:$ destroydb s-escolares
loltun:$
```

CREATE

```
create relación (atributo=tipo) [, (atributo=tipo)]
```

Una vez creada la base de datos, podemos crear las relaciones que se quieran con el nombre de la relación y parejas que definen los atributos y dominios que forman el encabezado de la relación. Este al igual que todos los comandos internos de Ingres acaban con un \g.

ejemplo:

```
* create alumnos (no-cta=c9, nombre=c50, ap-paterno=c50,
                  ap-materno=c50) \g
```

DESTROY

```
destroy relación
```

Este comando borra por completo una relación dentro de la base de datos.

ejemplo:

```
* destroy alumnos \g
Executing . . .

continue
*
```

APPEND

```
append [to] relación (atributo=valor [,atributo=valor])
[ WHERE cuantificador ]
```

Para poder añadir una *n-ada* a una relación, o bien añadir a otra relación una *n-ada* siempre y cuando se cumpla la condición del cuantificador.

ejemplo sobre añadir de manera sencilla una *n-ada* a una relación:

```
* append alumnos (no-cta="8315637-7",
* nombre="Victor Hugo",
* ap-paterno="Dorantes", ap-materno="Gonzalez") \g
Executing . . .

(1 tuple)

continue
*
```

ejemplo de como añadir información de una relación a otra:

```
* create carreras (no-cta=c9, carrera=c2) \g
Executing . . .

continue
* range of n is alumnos
* append to carreras (n.no-cta, carrera="24")
* where n.no-cta="8315637-7" \g
Executing . . .

(1 tuple)

continue
*
```

DELETE

delete variable [WHERE cuantificador]

Este comando permite borrar todo un conjunto de *n-adas* de una relación que cumplen con el cuantificador. Nótese que el comando actúa sobre una variable de la relación que se evalua en cada una de las n-adas, en caso de que todas las n-adas cumplan el cuantificador o éste se omita entonces dá como resultado una relación vacía.

ejemplo:

```
* range of n is carreras
* delete n
* where n.carrera="21" \g
Executing . . .

(547 tuples)

continue
*
```

REPLACE

replace variable (atributo=valor [, atributo=valor])
[WHERE cuantificador]

Para poder hacer modificaciones a los valores de ciertos atributos.

ejemplo que supone un error de captura en un atributo:

```
* range of n is alumnos\g
* replace n (ap-paterno="Dorantes")
* where n.no-cta="8315637-7" \g
Executing . . .
(1 tuples)
continue
*
```

RETRIEVE

retrieve [[into] relación] (lista_de_atributos)
[WHERE cuantificador]

Este es el comando destinado para hacer todo tipo de consultas a cualquier relación de la base de datos.

ejemplo:

```
* retrieve (e.no-cta, e.ap-paterno)
* where e.no-cta="8315637-7" \g
Executing . . .
|no-cta   |ap-paterno   |
|----------------------|
|8315637-7|Dorantes     |
|----------------------|
(1 tuple)
continue
*
```

otro ejemplo de consulta entre dos relaciones:

```
* range of e is alumnos
* range of n is carreras
* retrieve (n.carrera,e.ap-paterno,e.ap-materno,e.nombre)
* where e.no-cta=n.no-cta and e.ap-paterno="Dorantes" \g
Executing . . .
|carrera|ap-paterno  |ap-materno  |nombre          |
|-------------------------------------------------|
|24     |Dorantes    |Gonzalez    |Victor Hugo     |
|-------------------------------------------------|
(1 tuple)
continue
*
```

2.1.2 EQUEL

Para permitir el desarrollo de aplicaciones que no dependieran de la forma interactiva inicial que propone *INGRES*, se desarrolló un nuevo lenguage llamado **EQUEL** (Embedded **QUEL**) el cual consiste esencialmente del lenguage **C** y los comandos de **QUEL**. De esta forma es posible programar en *C* con la posibilidad de procesar cualquier consulta o manejo de las bases de datos dentro del programa. Con esto en mente, la definición del lenguaje es la siguiente:

1) Cualquier instrucción de **C** es una intrucción válida de **EQUEL**.

2) Cualquier instrucción de QUEL es una intrucción válida de **EQUEL** la cual debe ser precedida por los simbolos ##.

3) Las variables de **C** pueden ser usadas dentro de cualquier intrucción de QUEL. Las declaraciones de estas variables también deben ser precedidas por un par de #.

Por ejemplo: La intrucción *RETRIEVE* sin una relación resultado tienen la siguiente forma:

```
##RETRIEVE (lista_de_atributos)
  [WHERE cuantificador]
##{
  Bloque de instrucciones en C
##}
```

los resultados de la consulta son utilizados en el bloque de instrucciones, el cual se ejecuta una vez por cada *n-ada* del cuantificador.

El único problema que podemos manifestar en contra de **EQUEL** pero que en realidad tiene su justificación, es que no se deben mezclar los bloques de consulta en **QUEL** con llamadas recursivas.

Por otro lado el proceso de compilación y generación de aplicaciones que ofrece *INGRES* es el siguiente:

i) El archivo con el código en **EQUEL** que debe tener extensión **.q**, se preprocesa con un programa que precisamente se llama **equel**.

```
$ equel aplica.q
```

ii) El preprocesador genera un archivo con extensión **.c** y este se compila incluyendo el archivo de definición de funciones de INGRES y ligando la biblioteca de funciones de INGRES llamada **libq.a**

```
$ cc -o aplica aplica.c -I/usr/local/include
  -L/usr/local/lib -lq
```

el resultado es un programa ejecutable llamado `aplica` el cual llama directamente las instrucciones de INGRES, saltandose al proceso 1 en la estructura de procesos e interaccionando de una manera más directa con la base de datos.

2.2 mSQL

Este manejador de bases de datos tiene sus origenes en un proyecto llamado *Minerva Network Managment Environment*, desarrollado en 1993 en la universidad de Bond, en Australia. Inicialmente *Minerva* utilizaba a *Postgres* [1] y para que *Minerva* tuviera una interfaz o comunicación con otros manejadores comerciales se desarrolló un convertidor de los predicados de *SQL* a *PostQUEL* llamado precisamente mSQL; pero se encontraron con que *Postgres* mismo utilizaba ya demasiados recursos y soporte. Cuando requirieron de ganar velocidad y realizar varias operaciones en paralelo se encontraron que Postgres generaba una copia de él mismo por cada proceso, lo cual en espacio de memoria fue más allá de lo que esperaban. Por otro lado, cuando trataron de correr Postgres en una Silicon Graphics se encontraron que no había soporte para que éste funcionara en tal equipo. Estas fueron las principales razones que motivaron el desarrollo de Mini-SQL como un manejador de bases de datos.

Mini-SQL ofrece un subconjunto del estándar de SQL (de ahí que se llame mini-SQL) basado en las especlficaciones del SQL ANSI. El paquete comprende el manejador de bases de datos, un programa monitor, un programa de administración, y sobre todo una biblioteca de funciones para el lenguaje C que permité el desarrollo de aplicaciones.

mSQL ofrece un modelo de trabajo basado en un ambiente cliente–servidor sobre una red con protocolo TCP/IP, lo cual lo hace bastante interesante ya que permite hacer consultas desde un conjunto de nodos ajenos al servidor; básicamente, trabaja con un proceso de fondo (*daemon* o demonio) que realiza el trabajo del servidor estableciendo conexiones a través de *sockets*. Este proceso puede establecer multiples conecciones y recibir varias consultas las cuales procesa serialmente formando una lista.

El software completo puede ser obtenido vía *ftp anónimo* de *ftp.bond.edu.au* dentro del directorio */pub/Minerva/msql*. Existe una lista de discusión sobre el desarrollo

[1] *Postgres es un manejador desarrollado en la Universidad de California en Berkeley, se fundamenta en un modelo extendido del modelo relacional y ofrece un lenguaje de consulta que es un superconjunto de QUEL llamado PostQUEL*

y demás temas relacionados al manejador a la cual puede uno suscribirse escribiendo a la dirección *majordomo@bond.edu.au* con una linea que diga:

subscribe msql-list mi-email@mihost

2.2.1 El Subconjunto de SQL

El subconjuto de instrucciones de SQL que soporta **mSQL** son los siguientes:

CREATE TABLE

Esta instrucción se usa para crear tablas. SQL permite el manejo de llaves primarias pero tiene la limitación que defina sólo un atributo como llave primaria de la tabla.

```
CREATE TABLE nombre_tabla (
        nombre_col tipo_col [ not null | primary key]
        [, nombre_col tipo_col [ not null | primary key ]]**
    )
```

Un ejemplo es:

```
CREATE TABLE alumnos (
    nombre char(25) not null,
    ap_paterno  char(25) not null,
    ap_materno char(25) not null,
    no_cta  char(9) primary key
)
```

DROP TABLE

Esta cláusula sencillamente borra una tabla dentro de una base de datos y su forma general es:

```
DROP TABLE nombre_tabla
```

por ejemplo:

```
DROP TABLE alumnos
```

INSERT

A diferencia del *SQL-ANSI*, esta instrucción no necesita de un *select* para poder hacer la inserción de una *n-ada* y por otro lado la forma general nos dice que debes dar una lista con los nombres de los atributos o columnas por un lado y por otro la lista de valores asociados a cada atributo.

```
INSERT INTO nombre_tabla (columna [, columna])
        VALUES (valor [, valor])
```

por ejemplo:

```
INSERT INTO alumnos (nombre, ap_materno, ap_paterno, no_cta)
VALUES ('Victor Hugo', 'Dorantes', 'Gonzalez', '8315637-7')
```

DELETE

La instrucción para borrar *n-adas*, donde se puede observar que la condición puede ser una expresión compuesta de manera que se borre un conjunto de *n-adas*.

```
DELETE FROM nombre_tabla
  WHERE columna OPERADOR valor
    [AND | OR columna OPERADOR valor]**

OPERADOR: <, >, ≤, ≥, <>, =, o like
```

ejemplo:

```
DELETE FROM alumnos
WHERE no_cta = '8315637-7'
```

SELECT

Sin duda esta es la instrucción de mayor peso dentro de *SQL* y en este caso *mSQL* tiene la limitación de no manejar funciones implicitas como *count()*, *avg()* y otras mas.

```
SELECT [nombre_tabla.]columna [, [nombre_tabla.]columna]**
FROM nombre_tabla [, nombre_tabla]**
  [WHERE [nombre_tabla.]columna OPERADOR VALOR
    [AND | OR [nombre_tabla.]columna OPERADOR VALOR]**]
  [ORDER BY [nombre_tabla.]columna [DESC]
    [, nombre_tabla.]columna [DESC]]

OPERADOR = <, >, ≤, ≥, ≤≥, =, o like
VALOR puede ser un valor o el nombre de una columna
```

ejemplo:

```
SELECT nombre, ap_paterno, ap_materno, no_cta FROM alumnos
  WHERE ap_paterno='Gonzalez'
```

El poder real de este operador es cuando se hacen uniones de tablas para establecer relaciones lógicas entre ellas, por ejemplo:

```
SELECT alumnos.no_cta, carreras.nombre
  FROM alumnos, carreras
  WHERE alumnos.no_cta=carreras.no_cta
```

UPDATE

Este cláusula tiene como caraterística que el valor a asignar en una columna debe ser un valor literal y no permite el nombre de una columna como en un *SELECT*.

```
UPDATE nombre_tabla SET columna=valor [, columna=valor]**
  WHERE columna OPERADOR valor
```

ejemplo

```
UPDATE alumnos SET carrera=21
  WHERE no_cta='8315637-7'
```

2.2.2 La biblioteca de funciones de mSQL

Al igual que muchos manejadores de bases de datos, *mSQl* provee una biblioteca de funciones llamada *libmsql.a* la cual nos ofrece un conjunto de funciones con las cuales podemos comunicar programas en **C** con el manejador.

La manera en que trabajan el conjunto de funciones de esta biblioteca es esencialmente bajo el modelo cliente-servidor en que trabaja **mSQL**. Esto quiere decir que se establece una conección al servidor a través de un *socket* (lo cual tiene la ventaja de que el servidor puede estar sobre la red en cualquier otro lugar); una vez establecida esta conexión se le dice al servidor que base de datos se va a utilizar y se realizan las operaciones de consulta sobre la misma y, finalmente, se cierra la conexión.

Entre las funciones más importantes de la biblioteca de funciones de mSQL se encuentran las siguientes:

Función	Descripción
`msqlConnect ()`	Establece la conexión con el manejador
`msqlSelectDB ()`	Selecciona la base de datos
`msqlQuery ()`	Envia la consulta al manejador
`msqlNumRows ()`	El número de renglones de una consulta
`msqlFetchField ()`	El número de columnas
`msqlListDBs ()`	Una lista de las bases de datos existentes
`msqlListTables ()`	Una lista de tablas o relaciones en la base de datos
`msqlClose ()`	Cerrar la conexión con el manejador

Capítulo 3

Tcl/Tk

3.1 Tcl

Tcl es el acrónimo de *Tool Command Language*, un lenguaje de programación e interacción, de los llamados *scripts*; *i.e.* un lenguaje a nivel de un shell para *UNIX* del estilo *csh*, *sh*, *ksh*, *bash*, etcétera. Fue desarrollado en la Universidad de California en Berkeley en 1993 por el Dr. John K. Ousterhout. La primera versión oficial liberada fue la 7.3 en *Tcl* y la 3.6 en su extensión *Tk* (toolkit); actualmente se distribuyen las versiones 7.5 y 4.1, respectivamente.

En realidad *Tcl* es dos cosas, un lenguaje de comandos y una biblioteca de funciones que permite desarrollar extensiones al conjunto básico de comandos. El lenguaje *Tcl* al igual que un shell de UNIX está orientado a que sus instruccciones sean llamadas a otros programas y a través de este se puede interactuar con programas, editores, depuradores y shells. La sintaxis de sus comandos es muy sencilla y permite crear procedimientos que aumentan la capacidad de sus propios comandos.

Al ser un lenguaje al nivel de un shell podemos desarrollar aplicaciones orientadas a control de procesos, administración del sistema, presentación de resultados, etcétera. Si ha esto último añadimos *Tk* el cual es una extensión de Tcl orientada al desarrollo de aplicaciones en modo gráfico entonces podemos apreciar aún más que se tienen las herramientas para diseñar interfaces gráficas a aplicaciones de todo tipo.

3.1.1 Estructuras generales

Un programa en *Tcl* es conocido como un *script*, el cual consiste de un conjunto de comandos separados por un";" o por un cambio de linea. *Tcl* posee una sola estructura sintáctica para todos sus comandos, la cual es:

comando *arg1 arg2 ...*

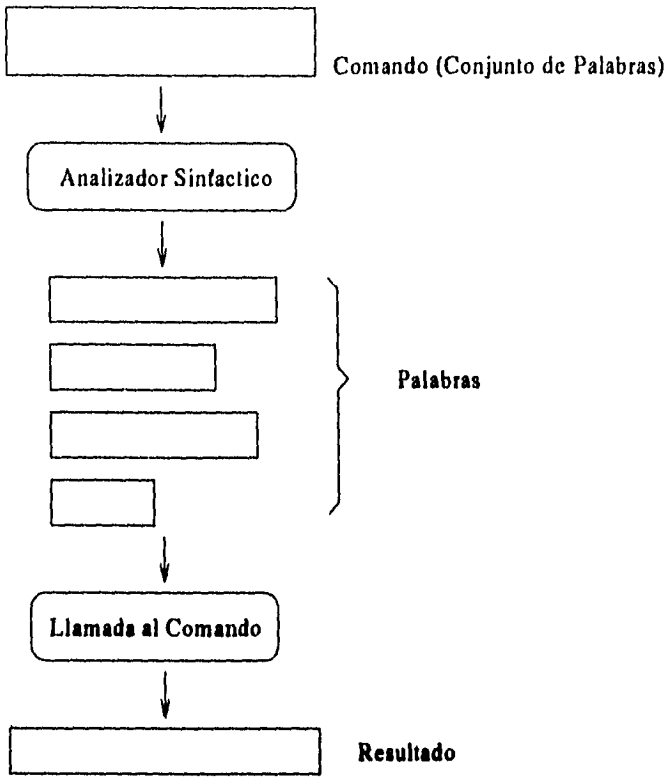


Figura 3.1 El analizador sintáctico de Tcl

por lo que, un comando consiste de una o más palabras donde la primera de ellas es el nombre del comando y las palabras restantes son los argumentos del comando. Los argumentos van separados por un espacio en blanco o un tabulador.

La forma en que el intérprete de *Tcl* analiza los comandos se basa en la figura 3.1, el intérprete procesa un comando en dos fases, en la primera se realiza el análisis sintáctico (*parseo*) de la cadena original para producir una secuencia de palabras separadas. La segunda fase consiste en checar si con respecto a la primera palabra que se obtuvo existe un comando que se llame así. De existir, se manda ejecutar pasándole como argumentos el resto de la cadena analizada. Todos los comandos producen un resultado (aunque sea una cadena vacia).

Este esquema, junto con tres formas de sustitución permiten la composición y evaluación de comandos. La primera de ellas es llamada ***sustitución de variables***, esta se obtiene anteponiendo a una variable el signo de dollar (**$**), para de esta manera obtener el valor contenido en la variable.

Ejemplo:

```
set edad 28
expr $edad*12
```

en este caso el comando **set** toma dos argumentos, el primero es el nombre de una variable y el segundo el valor a asignarle. El comando expr toma un solo argumento (una expresión) y lo evalúa para regresar el resultado de tal evaluación. Aquí podemos apreciar la sustitución de la variable; **$edad** denota *el valor* de la variable.

El segundo tipo de sustitución es llamado *sustitución de comandos*; esta ocasiona que un argumento de un comando sea reemplazado por el resultado de evaluar ese argumento como un comando de *Tcl*. Para hacer esta sustitución se encierra el comando de *Tcl* entre corchetes. Ejemplo:

```
set edad 28
set nmeses [expr $edad*12]
```

como vimos ya, el segundo argumento de **set** debe ser un valor y precisamente el resultado del comando **expr** al evaluar una expresión es un valor. De esta forma, antes de hacer la asignación a la variable se realiza la evaluación o sustitución de este argumento.

La tercera de las posibles sustituciones es la ***sustitución con backslash*** la cual es usada para insertar caracteres especiales como el cambio de línea, espacio en blanco, tabulador, corchetes, el signo **$** y otros. Ejemplo[†]:

```
   set mesg "El numero de meses es:  \n \t$nmeses"
⇒ El numero de meses es:
⇒ 336
```

Tcl provee varios mecanismos para prevenir al parser de interpretar ciertos caracteres, estas técnicas se les conocen como ***encomillar***; las comillas dobles deshabilitan que el contenido sea separado en multiples palabras. Ejemplo:

```
   set mesg "El numero de meses es:\n \t$nmeses"
⇒ El numero de meses es:
⇒          336
```

Otro mecanismo más radical es utilizando llaves ({}) dentro de las cuales todos los caracteres especiales pierden su significado. Ejemplo:

```
   set mesg {El numero de meses es:  \n \t$nmeses}
⇒ El numero de meses es:  \n \t$nmeses
```

[†] *En adelante se utilizará el simbolo ⇒ para hacer referencia a la respuesta o resultado que genera el intérprete.*

sin embargo la importancia de las llaves radica en realidad en lo que se conoce como *evaluación diferida*, esto es que al no ser procesada la cadena dentro de las llaves esta puede ser pasada al procedimiento de un comando como parte de sus argumentos. El procedimiento entonces podrá encargarse de evaluar los caracteres especiales por su cuenta. Esta estructura permite que los procedimientos, condicionales y ciclos puedan ser vistos en términos de la definición inicial de un comando en *Tcl*:

`if` $cond_1$ $bloque_1$ `?elseif` $cond_2$ $bloque_2$ ... `?else` $bloque_n$

`while` *cond bloque*

`for` *inicial cond incremento bloque*

`proc` *nombre args bloque*

Tcl tiene como única estrutura de datos básica las cadenas. Apartir de estas se definen dos estructuras de "alto nivel" como son las listas y los arreglos. Como es de esperar ambos se implementan a través de cadenas.

3.1.2 Listas

Una lista en Tcl es una colección de cualquier número de elementos separados por espacios o tabuladores. De acuerdo con las formas de sustitución que ya se vieron, resulta mas conveniente utilizar las llaves al asignar a una variable una lista de la forma:

```
   set lista { esta es una lista }
⇒ esta es una lista
   set $lista
⇒ esta es una lista
```

El conjunto de operaciones nativas de Tcl para listas son las siguientes:

Construir Listas

`list` *arg1 arg2 ...*
: Crea una lista con los argumentos de la función

`lappend` *lista arg1 arg2 ...*
: Añade al final de la lista los elementos que se señalan.

`concat` *lista1 lista2 ...*
: Concatena multiples listas produciendo una sola.

Accesar los elementos de una lista

`lindex` *lista i*
: Regresa el elemento que está en la posición i, considerando que al igual que en **C**, los elementos van de $0 \ldots n-1$

`llength` *lista*

Regresa la longitud o número de elementos de la lista.

`lrange` *lista i j*

Regresa una lista que empieza a partir del elemento i y termina en el elemento j.

Modificar listas

`linsert` *lista indice arg1 arg2 ...*

Inserta todo un conjunto de elementos en la lista a partir de un cierto indice.

`lreplace` *list i j arg1 arg2 ...*

Reemplaza los elementos que están entre los índices i y *j* por los argumentos de la función.

Búsquedas y ordenamientos

`lsearch` *list valor*

Regresa el índice del elemento en la lista que concuerda con el valor de acuerdo al modo especificado.

`lsort` *opciones lista*

Ordena los elementos de la lista de acuerdo a las opciones, entre las cuales estan `-increasing`, `-decreasing`.

3.1.3 Estructuras de control

Como se definió con anterioridad, las estructuras de control son también comandos de Tcl con un manejo especial. Las estructuras de control que posee Tcl son: **if, while, for, foreach, swith**.

Condicionales

`if` *expresion* `then` *bloque1* `?else` *bloque2*

Si bien las palabras `then` y `else` se pueden omitir, el `else` es altamente recomendable. El `if` recibe por lo menos dos argumentos que son la condición y el bloque a ejecutar en caso de sea cierta la condición.

Ejemplo:

```
if { $a >= $b} {
    set mayor $a
} else {
    set mayor $b
}
```

`switch` *opciones valor* `patron1` *bloque1* `patron2` *bloque2* ...

Switch se utiliza para una comparación del valor con los patrones definidos en una lista de pares patrón-bloque, solo el primer bloque del patrón que cumple la comparación es ejecutado.

Ejemplo:

```
switch $valor {
    a { incr t1 }
    b { incr t2 }
    c { incr t3 }
}
```

Ciclos

`while` ***condicion*** **`bloque`**

Se evalua la condición como una expresión. Si el valor es diferente de cero se evalua el bloque como un script mas de tcl y se vuelve a evaluar la condición.

Ejemplo:

```
set lista { hola a todo mundo }
set i $0
while { i < [llength $lista] } {
    puts [lindex $lista $i]
    incr i 1
}
```

`for` ***inicial condicion incremento bloque***

El `for` toma cuatro argumentos, siguiendo la forma del `for` de C, *i.e.* el primer argumento contiene todas las inicializaciones que queramos, una condición de evaluación y los incrementos, en caso de cumplirse la condición se evalua el bloque como un script mas de Tcl.

Ejemplo:

```
for { set i 0 ; set lista { hola a todo mundo } } {
    $i < [ llength $lista ] } { incr i } {
    puts [lindex $lista $i]
}
```

`foreach` *var lista bloque*

Este ciclo considera una variable que va tomando valores sobre la lista dada y por cada uno se ejecuta el bloque de comandos.

Ejemplo

```
foreach i { hola a todo mundo } {
    puts $i
}
```

3.1.4 Procedimientos y funciones

De acuerdo a la definición de Tcl, un procedimiento es un comando que tiene dos argumentos que son una lista con los parámetros y el cuerpo del procedimiento. Como en cualquier lenguaje de programación, uno define un conjunto de procedimientos los cuales más adelante se usan como comandos extras. Un procedimiento es creado con el comando proc con la siguiente sintáxis:

proc *nombre argumentos cuerpo*

ejemplo

```
proc sucesor { a } { incr a }
```

una vez definido el procedimiento basta llamarlo con el nombre y sus argumentos:

```
sucesor 43
⇒ 44
```

Como se puede observar, Tcl sigue la idea de C en el sentido que el procedimiento o función regresa como valor (de no ser explícito) el valor del último comando evaluado. La forma de hacer explícito el valor de la función es a través del comando **return**.

Todas las variables son locales a la función y si se necesita hacer explícito que algunas variables sean globales a través del comando **global variable**.

3.1.5 Procesos, manejo de errores y excepciones

Sin duda que es imposible considerar de interés un lenguaje script que no tenga una manejo de procesos y alguna forma de controlar las situaciones de error. Tcl posee un conjunto de comandos para interactuar con procesos para poder sacar provecho de trabajar en un ambiente como UNIX.

El comando `exec` permite ejecutar comandos o programas de UNIX desde un script de Tcl, ejemplo:

```
set d [exec date ]
```

en este caso la salida estándar del proceso ejecutado es el valor que se le asigna a la variable. Este comando soporta un conjunto de operadores de redireccionamiento y concatenación de procesos. Ejemplo:

```
set n [ exec sort < /etc/passwd | uniq | wc -l 2> /dev/null ]
```

en el caso de usar el operador **&** el porceso se ejecuta como un proceso de fondo y el comando exec regresa el identificador asignado al proceso.

Cuando un error ocurre en un programa script, normalmente ocasiona que se detenga la ejecución. Los errores sencillos son cuando no se cumple con la forma sintáctica del comando, pero el tipo de errores que nos interesa son aquellos en los cuales tenemos comandos que pueden fallar. Por ejemplo cuando se usan los elementos de una lista y esta lista es vacia.

Una forma de anticipar los errores es a través del comando **catch**, el cual nos permite continuar con la ejecución a pesar de que un error haya ocurrido. Ejemplo:

```
   catch [ unset x ]
⇒ 1
```

en este ejemplo el comando **unset** falla cuando se "libera" una variable que no existía previamente, entonces la función **catch** regresa un 0 en caso de que el comando se haya llevado a cabo con éxito y un valor distInto de 0 en caso contrario. De hecho si se le añade una variable como un argumento más, en esta regresa un mensaje de porque fracasó el comando.

El comando **catch** permite determinar si alguna parte crítica de nuestro script se llevó a cabo con éxito y el permitir continuar nos dá el mecanismo de manejar tales excepciones. Este mecanismo tan sencillo permite hacer sistemas suficientemente robustos y estables aún cuando éstos dependan de muchas condiciones externas e imprevistas.

3.2 Tk

Tk es un conjunto de herramientas (*toolkit*) para el desarrollo de interfaces gráficas para sistemas basados en X-Window. Tk extiende el conjunto de comandos que provee Tcl y agrega un conjunto de comandos para crear elementos llamados *widgets* (o ventanas). El tipo de ventanas que establece **Tk** está basado en el estándar *Motif*.

Las ventanas que define Tk se organizan de la misma manera que las ventanas de X-Window,*i.e.* a través de una jerarquía, la cual determina el nombre de las ventanas. Cada ventana definida por Tk cae dentro de una clase la cual determina su apariencia, comportamiento y uso; mientras que un manejador de geometrías controla su tamaño y localización sobre la pantalla.

3.2.1 Categorías de ventanas

Tk define 15 clases o categorias de ventanas: *Frame*, *Toplevel*, *Label*, *Message*, *Button*, *Checkbutton*, *RadioButton*, *Listbox*, *Scrollbar*, *Scale*, *Menu*, *Menubutton*, *Entry*, *Text*, *Canvas*. Todas ellas poseen un conjunto de atributos variables (algunos bastante grandes).

Frame

Una ventana de tipo frame es la más sencilla que puede haber, aparece en forma rectangular y puede tener bordes en 3D. Típicamente este tipo de ventana se utiliza para contener otras ventanas y agruparlas. Ejemplo:

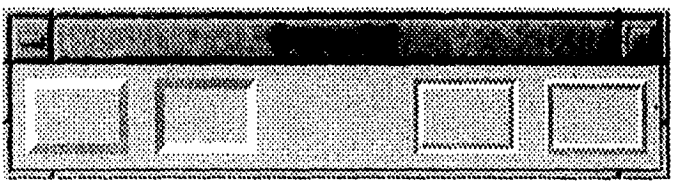


Figura 3.2. Frames

```
foreach relieve { raised sunken flat groove ridge } {
    frame .$relieve -width 15m -height 10m -relief $relieve \
    -borderwidth 4
    pack .$relieve -side left -padx 2m -pady 2m
}
```

Toplevel

Una ventana toplevel es idéntica a una de tipo frame, excepto porque ésta es una ventana independiente (llamada de alto nivel) pero que sigue estando dentro de la jerarquía. Típicamente se usan para generar paneles de control o ventanas de diálogos. Ejemplo:

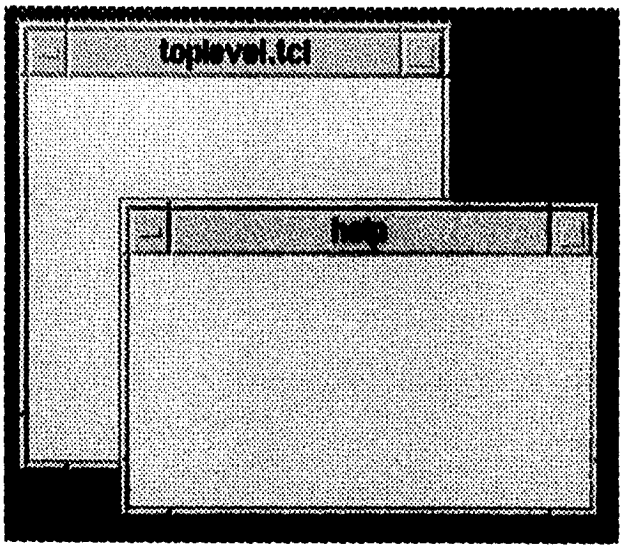


Figura 3.3 Toplevel

```
toplevel .help -height 100 -width 200
```

Label

Similar a una frame excepto porque puede desplegar una línea de texto o contener una imagen (de tipo bitmap). Ejemplo:

Figura 3.4 Label

```
label .hola -text "hola a todo mundo" -relief groove
pack .hola
```

Message

La única diferiencia con respecto a *label* es que puede desplegar multiples líneas de texto, puede romper la línea de acuerdo al tamaño de la ventana o su justificación. Ejemplo:

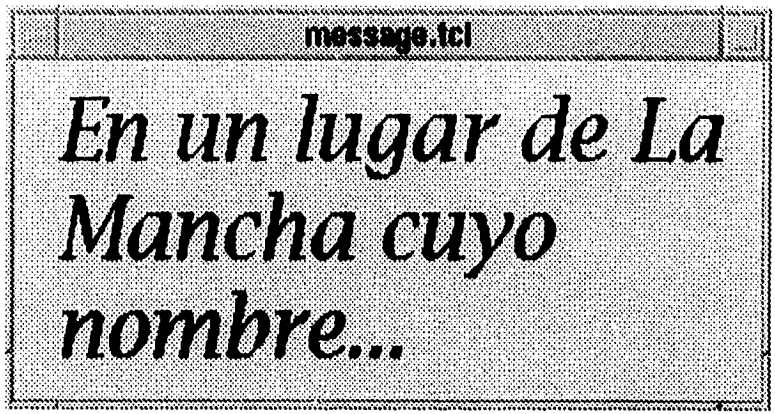


Figura 3.5. Message

```
message .mesg -width 5c -justify left \
    -text "En un lugar de La Mancha cuyo nombre..."
pack .mesg
```

Button

Esta categoría de ventana puede contener una cadena o una bitmap y esta ligada a una acción del ratón (un click sobre ella), tiene un efecto de ***presionado*** y puede tener asociado a tal evento la ejecusión de un conjunto de comandos. Ejemplo:

Figura 3.6. Button

```
button .salir -text "Salir" -command exit \
  -command quit
pack .salir
```

Checkbutton

Al igual que *button*, está asociado a un click sobre la ventana pero con la característica de que tiene asociada una variable. De esta forma se pueden combinar varios para seleccionar entre un conjunto de atributos dentro de una aplicación. Ejemplo:

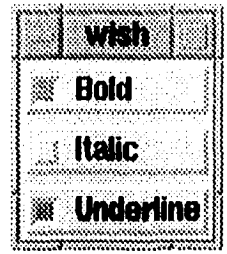


Figura 3.7. Checkbutton

```
checkbutton .bold -text Bold -variable bold -anchor w
    -relief raised
checkbutton .italic -text Italic -variable italic -anchor w
    -relief raised
checkbutton .underline -text Underline -variable under
    -anchor w -relief raised
pack .bold .italic .underline -side top -fill x
```

Radiobutton

A diferencia del *checkbutton*, un conjunto de *radiobuttons* están asociados a una sola variable y sólo uno de estos botones puede ser seleccionado a la vez. Ejemplo:

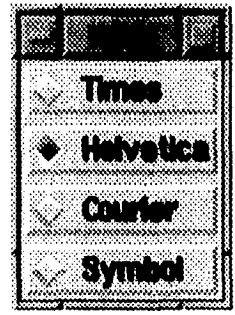


Figura 3.8. Radiobutton

```
radiobutton .times -text Times -variable font -value times \
    -anchor w -relief raised
radiobutton .helvetica -text Helvetica -variable font \
    -value helvetica -anchor w -relief raised
radiobutton .courier -text Courier -variable font \
```

```
    -value courier -anchor w -relief raised
radiobutton .symbol -text Symbol -variable font -value symbol \
    -ancho w -relief raised
pack .times .helvetica .courier .symbol -side top -fill x
```

Listboxes

Esta categoría de ventana tiene asociada una lista de valores que pueden ser incorporados en la ventana (uno por línea) y hacer selcciones sobre ella. Normalmente se usa en combinación con ventanas de tipo *scrollbar* para poder contener listas más grandes de las que pueden mostrarse.

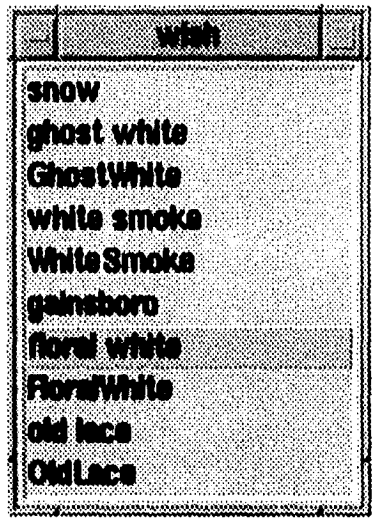


Figura 3.9. Listboxes

```
listbox .colores
pack .colores
set f [open /usr/lib/X11/rgb.txt]
while { [gets $f line] >= 0 } {
    .colores insert end [lrange $line 3 end]
}
close $f
```

Scrollbar

Este tipo de ventanas se usan en combinación con *Listboxes*, *Textbox* o *Canvas* para hacer desplazamientos o movimientos de los elementos de las otras ventanas en forma horizontal o vertical. Consiste de flechas fijas a los extremos (que permiten movimientos cortos) y una barra al centro (*slider*) que permite desplazamientos arbitrarios con ayuda del ratón. Ejemplo:

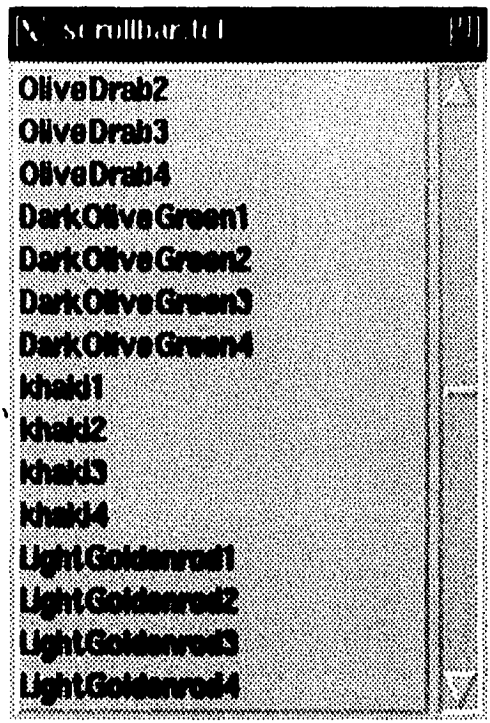


Figura 3.10. Scrollbar

```
listbox .colores -borderwidth 2 -relief raised \
    -yscrollcommand ".scroll set"
pack .colores -side left
set f [open /usr/lib/X11/rgb.txt]
while { [gets $f line] >= 0 } {
.colores insert end [lrange $line 3 end]
}
close $f
scrollbar .scroll -command ".colores yview"
pack .scroll -side right -fill y
```

Scales

Una ventana de este tipo permite manejar y seleccionar escalas a través de un boton de desplazamiento que se mueve a través de un rango o escala definida. Permite ejecutar un conjunto de comandos por cada valor que toma dentro de la escala, ejecutar un conjunto de comandos. Ejemplo:

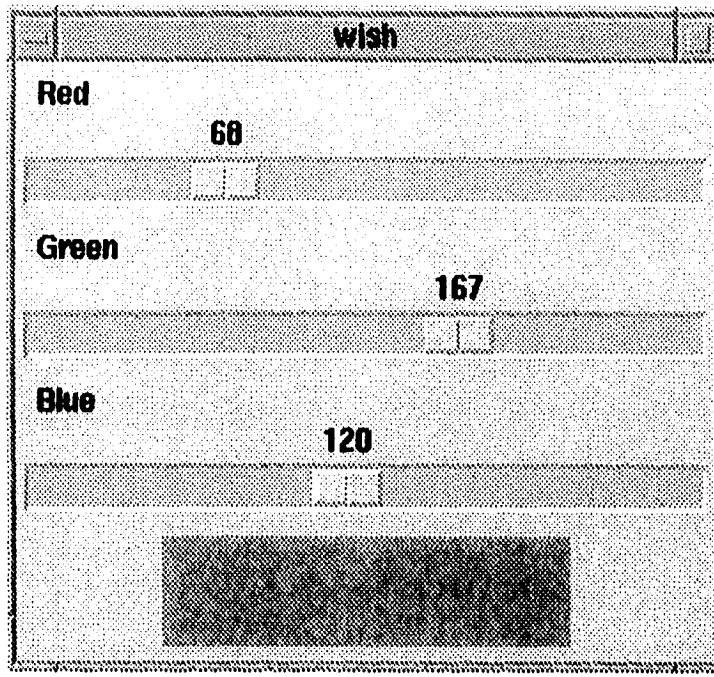


Figura 3.11. Scalebox

```
scale .red -label Red -from 0 -to 255 -length 10c \
    -orient horizontal -command newColor
scale .green -label Green -from 0 -to 255 -length 10c \
    -orient horizontal -command newColor
scale .blue -label Blue -from 0 -to 255 -length 10c \
    -orient horizontal -command newColor
frame .sample -height 1.5c -width 6c
pack .red .green .blue -side top
pack .sample -side bottom -pady 2m
proc newColor value {
    set color [format #%02x%02x%02x \
    [.red get] [.green get] [.blue get]]
    .sample configure -background $color
}
```

Menús y Menubuttons

Una ventana de tipo menú es básicamente un esqueleto para formar menús a través de montar menubuttons, estos menus pueden ser de tipo *pull-down*, de cascada y *pop-up*. Este tipo de ventana contiene un arreglo de entradas en forma de columna en donde se pueden insertar diferentes tipos de botones como son: radiobutton, checkbutton, cascade, separador y boton de comandos.

Un menubutton forma parte de un *Menú*, y contiene a su vez un conjunto de ventanas de tipo *label*, *checkbutton* o *radiobutton*. Ejemplo:

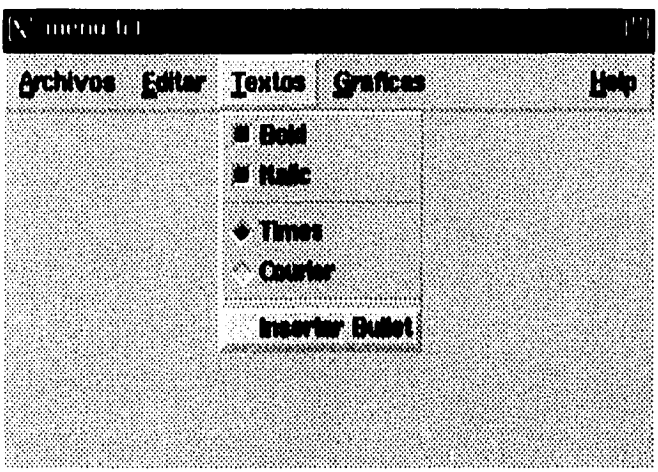


Figura 3.12. Menu

```
frame .mbar -relief raised -bd 2
frame .caja -width 10c -height 5c
pack .mbar .caja -side top -fill x
menubutton .mbar.file -text Archivos -underline 0 \
    -menu .mbar.file.menu
menubutton .mbar.edit -text Editar -underline 0 \
    -menu .mbar.edit.menu
menubutton .mbar.graf -text Graficas -underline 0 \
    -menu .mbar.graf.menu
menubutton .mbar.text -text Textos -underline 0 \
    -menu .mbar.text.menu
menu .mbar.text.menu
.mbar.text.menu add checkbutton -label Bold -variable bold
.mbar.text.menu add checkbutton -label Italic -variable italic
.mbar.text.menu add separator
```

```
.mbar.text.menu add radiobutton -label Times -variable font \
    -value times
.mbar.text.menu add radiobutton -label Courier -variable font \
    -value courier
.mbar.text.menu add separator
.mbar.text.menu add command -label "Insertar Bullet" \
    -command "insertarBullet"
menubutton .mbar.help -text Help -underline 0 \
    -menu .mbar.help.menu

pack .mbar.file .mbar.edit .mbar.text .mbar.graf -side left
pack .mbar.help -side right -fill x
```

Entries

Esta ventana permite la entrada o captura de datos a una aplicación, se define una área de inserción y es posible asociarle una variable para accesar el valor de entrada. Ejemplo:

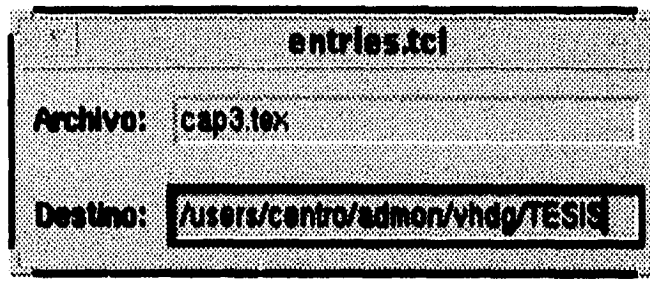


Figura 3.13. Entrybox

```
frame .arch
label .arch.label -text "Archivo:"
entry .arch.dato -width 28 -relief sunken -bd 2 \
    -textvariable nombre

pack .arch.label .arch.dato -side left -padx 1m -pady 2m
frame .dest
label .dest.label -text "Destino:"
entry .dest.dato -width 28 -relief sunken -bd 2 \
    -textvariable dir

pack .dest.label .dest.dato -side left -padx 1m -pady 2m
pack .arch .dest -side top
```

Text

Como su nombre lo dice, este tipo de ventana permite manejar multiples líneas de texto y asociarla con un scrollbar para darle movimiento al texto contenido, es muy útil para editar o modificar archivos además de poder asociar *ligas* parte del texto para asociarla con alguna acción. Ejemplo:

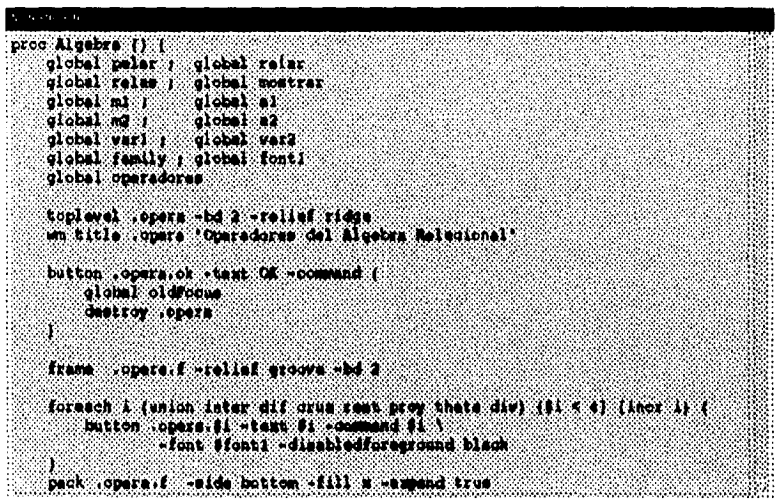


Figura 3.14. Textbox

```
text .text -relief raised -bd 2 \
    -yscrollcommand ".scroll set"
scrollbar .scroll -command ".text yview"
pack .scroll -side right -fill y
pack .text -side left
proc loadFile file {
    .text delete 1.0 end
    set f [ open $file ]
    while { ![eof $f] } {
      .text insert end [ read $f 1000 ]
    }
    close $f
}
loadFile ~TESIS/AlgTk/Alg-Tk.tcl
```

Canvas

Esta es la única categoría que permite primitivas para poder dibujar y hacer trazos geométricos, insertar iconos, montar otras ventanas y sobre es capaz de mantener una relación de los elementos que contiene para poder interactuar con ellos. Ejemplo:

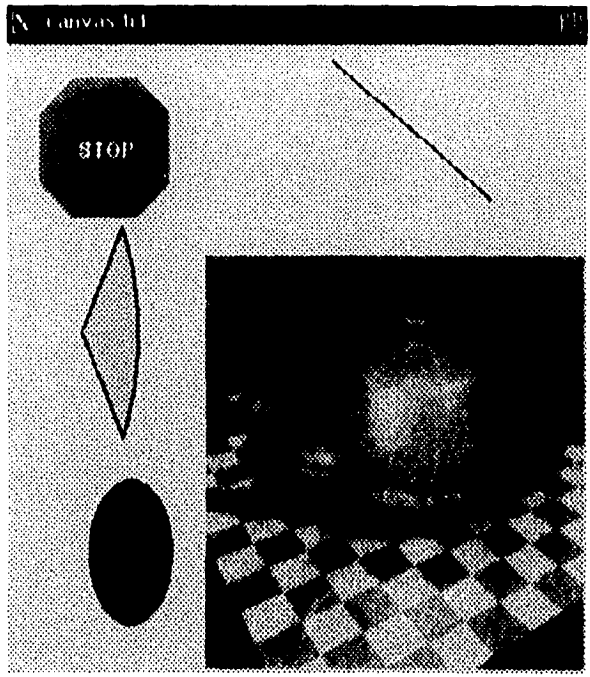


Figura 3.15. Canvas

```
canvas .canv -width 10c -height 10c
pack .canv
.canv create line 200 1 300 90
.canv create arc 10 80 80 250 -start 45 -extent -90 \
    -style pieslice -fill orange -outline black
.canv create oval 50 250 100 330 -fill blue -width 0
.canv create poly 80 20 100 40 100 80 80 100 40 100 \
    20 80 20 40 40 20 -fill red
.canv create text 60 62 -text STOP -fill white
image create photo p1 \
    -file /usr/local/lib/tk/demos/images/teapot.ppm
.canv create image 250 250 -image p1
```

3.2.2 El manejador de geometría Packer

Las ventanas viven dentro de una jerarquía y el manejo de tal jerarquía y otras funciones no son propias de ellas sino de lo que se conoce como un *manejador de geometría*. Entre las funciones más importantes de un manejador de geometría es controlar la posición y el tamaño de las ventanas, además a través de él es como aparecen o desaparecen las ventanas de la pantalla.

Este esquema determina que pueden existir varios manejadores de geometrías y básicamente la diferencia entre uno y otro reside en los algoritmos o mecanismos con los que se acomodan las ventanas en la pantalla. Tk tiene dos manejadores que son el *Placer* y el *Packer*, de estos dos el Placer es un manejador orientado a coordenadas dentro de las ventanas mientras que el Packer es un manejador de geometría más poderoso y que utiliza un algoritmo sencillo y claro para acomodar las ventanas sin necesidad de dar muchos detalles sobre las ventanas y sus coordenadas.

El Packer se basa en la jerarquía de las ventanas por lo que su algoritmo esta basado en ubicar a las ventanas hijas dentro del espacio de la ventana padre, para esto cada ventana padre tiene asignado un espacio y el Packer tiene que insertar y distribuir en ese espacio a las ventanas hijas. El Packer entonces puede acomodar a las ventanas hijas poniendolas todas en forma de un arreglo a la izquierda o derecha (en forma de renglón) o bien un arreglo de arriba hacia abajo o de abajo hacia arriba (en forma de columna), e incluso es posible mezclar entonces los renglones y columnas.

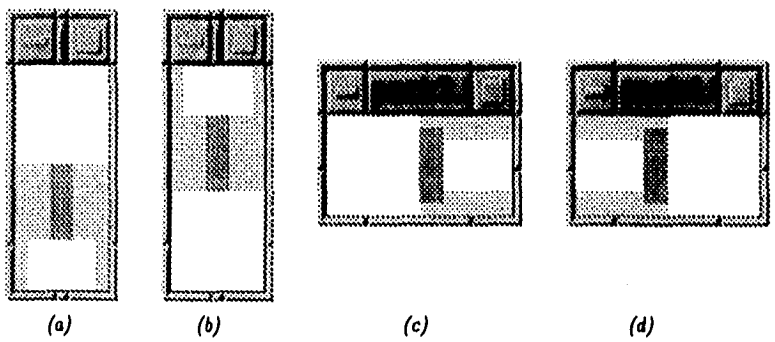


Figura 3.16 El manejador de geometría Packer

Como se vió en los ejemplos de las categorías de las ventanas el comando **pack** es con el cual se interactúa con el Packer, y de estos ejemplos anteriores la única diferencia entre uno y otro fue la opción **-side** que le indica al Packer la orientación en la cual va a acomodar las ventanas. Esta opción puede ser (a) `top`, (b) `bottom`, (c) `left` o (d) `right`, lo que normalmente se estila para poder hacer interfces que tengan

una estructura muy compleja es utilizar las ventanas de tipo frame para albergar a las otras ventanas y tener un ordenamiento dentro de estos frames mas sencillo y claro.

3.2.3 Eventos

Una vez diseñada una interfaz gráfica para alguna aplicación ésta interactúa con el usuario a través de lo que se conoce como *eventos*, estos son básicamente las acciones que sobre el ratón o teclado se hacen. Prácticamente cualquier movimiento o presión de alguno de los botones es un evento, Tk proporciona su comando **bind** con cual es posible relacionar alguno de estos eventos a una ventana y poder ejecutar alguna accción. De hecho este esquema de trabajo se manifiesta en la estructura misma del comando como se ve a continuación:

bind . *ventana evento* { *accción* }

Existe todo un conjunto de los posibles eventos los cuales están asociados tanto al ratón como al teclado y son eventos que van desde si el ratón se movió, cual de los botones fue presionado, cuantas veces se precionó algún botón, cuál es la posición actual del ratón, sobre cuál de las ventanas de la interfaz se encuentra el ratón, cualquier tecla o combinación de ellas, etcétera. Ejemplos:

```
bind all <Leave> { puts "Saliendo de la ventana %W "}
bind all <Motion> { puts "El ratón esta en la pos:  (%x, %y) "}
```

Al tipo de programación en interfaces gráficas también se le conoce como programación orientada a eventos y Tk con este mecanismo de comunicación entre la aplicación y los eventos permite tener gran control sobre lo que el usuario trata de realizar sobre la aplicación.

Capítulo 4

Alg

4.1 Ideas principales

Tcl ha sido diseñado para que el conjunto básico de comandos del lenguaje sea extendido fácilmente. Si bien existen comandos con los cuales es posible interactuar con programas externos, como lo es un manejador de bases de datos, la ventaja de tener un mayor control por parte de una aplicación que use esta extensión, sobre la comunicación con tales programas suele ser razón suficiente para optar por crear una extensión de Tcl responsable de tal interfaz.

Dicha extensión se lleva a cabo utilizando la biblioteca de funciones de Tcl, en base al esquema se muestra en la siguiente figura.

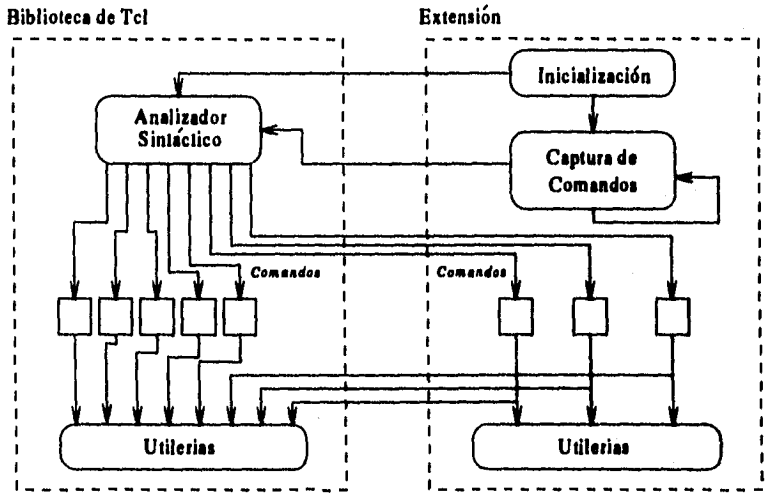


Figura 4.1 Esquema de extensión de Tcl

El esquema anterior plantea una forma de trabajo muy parecida al esquema presentado en la sección 3.1.1; muestra el módulo que proporciona Tcl, que consta de

un analizador sintáctico el cual llama a los comandos propios de Tcl, o bien aquellos que pertenecen a la extensión, y estos llaman a su vez al conjunto de utilerias o código a ejecutar.

La parte de la extensión consiste de una pequeña parte de inicialización en la cual se dan de alta las estructuras propias y los nuevos comandos, los cuales aparecen en la parte baja del módulo de la extensión dentro de la figura, y un ciclo de captura de comandos propio de un intérprete.

Con esto, se tiene que la extensión plantea la creación de un nuevo interprete, el cual, junto con los comandos que se le incorporan, le dan una orientación particular a Tcl. Esta es una de la características que ha hecho tan atractivo el uso de Tcl y que permite a prácticamente cualquier área del cómputo utilizar el conjunto de comandos de Tcl y sólo preocuparse por cuales deben ser los nuevos comandos y su implementación, para que este nuevo interprete proporcione todas las herramientas para generar aplicaciones.

Con esto se crea un comando llamado `alg` que contiene toda una categoria de comandos llamado ALG, *al nivel de los propios de Tcl*, que estén orientados enteramente a nuestra aplicación. El esquema de desarrollo se muestra en la siguiente figura:

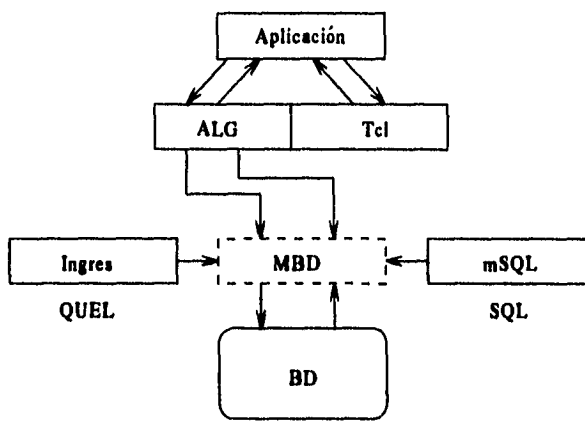


Figura 4.2. Esquema de la extensión Alg

El trabajo realizado consistió en extender Tcl, desarrollando comandos que emulan los operadores del álgebra relacional. De esta manera es posible desarrollar aplicaciones basadas directamente en los operadores del álgebra relacional sin que ninguna aplicación tenga que ver con cual manejador se esta usando o cual es su lenguaje de manipulación de datos.

Para poder distinguir los comandos de la extensión de los nativos de Tcl, éstos tendrán un prefijo llamado `alg`, de manera que la forma general de los comandos será: `alg comando parámetros`.

4.2 Extensión de Tcl

La biblioteca de Tcl posee todo un conjunto de funciones las cuales pueden ser clasificadas de acuerdo a la tarea que realizan. Por ahora nos ocuparemos de aquellas que son básicas para definir y crear un nuevo interprete: **Tcl_Main**, **Tcl_AppInit** y **Tcl_Init**. Estas funciones se usan dentro de la función principal en C de la extensión, que tiene la siguiente forma:

```
#include <tcl.h>

main (int argc, char *argv[]) {
    Tcl_Main (argc, argv, Tcl_AppInit) ;
    exit (0) ;
}

int Tcl_AppInit (Tcl_Interp *interp) {
    if (Tcl_Init (interp) == TCL_ERROR) {
      return TCL_ERROR ;
    }

    if (Alg_Init (interp) == TCL_ERROR) {
      return TCL_ERROR ;
    }

    return TCL_OK ;
}
```

La función **Tcl_Main** básicamente crea una nueva estructura de tipo **Tcl_Interp** la cual contiene estructuras sobre todo para administrar el resultado de las funciones de la biblioteca y en su caso las cadenas con los mensajes de error. **Tcl_Main** recibe como uno de sus parámetros el nombre de la función que se encargará tanto de la inicialización del conjunto de comandos básicos de Tcl como de los comandos de la extensión; como se observa en el código, ésta es llamada **Tcl_AppInit**.

Como parte del código de la función **Tcl_AppInit** se hace la llamada a la función **Tcl_Init** la cual se encarga de crear todos los comandos nativos y todas las estructuras de Tcl. La función **Alg_Init** será la función dentro de la cual serán creados

los comandos de la extensión usando otra función de la biblioteca de Tcl llamada Tcl_CreateCommand como se muestra a continuación:

```
int Alg_Init (Tcl_Interp *interp) {

    Tcl_CreateCommand (interp, "alg", tcl_alg,
                       (ClientData)0, (Tcl_CmdDeleteProc *) 0) ;

    return  TCL_OK ;
}
```

Nuestra extensión crea sólo un nuevo comando, llamado **alg**, pero éste contendrá todo el conjunto de operadores de la extensión. Como se observa, además esta función asocia al nuevo comando una función, misma que será ejecutada con cada llamada a este nuevo comando junto con todos los parámetros con que sea llamado. Esta función, llamada **tcl_alg**, tiene dos arreglos, uno con los nombres de los comandos y otro con el nombre de las funciones correspondientes a tales comandos. El código de esta función se ve así:

```
static int Tcl_alg (ClientData client, Tcl_Interp *interp,
                    int argc, char **argv)
{
    int i ;
    char msg[ 80 ] ;

/* Los comandos de la extensión */
    static char *cmds[] = {
        "abrebd",
        "cierrabd",
            ..
            ..
            ..
        0
    } ;

/* Las funciones de cada comandos */
    static (*f[])() = {
        Tcl_alg_AbreBD,
        Tcl_alg_CierraBD,
            ..
            ..
            ..
    } ;
```

```
    for (i = 0 ; cmds[ i ] != NULL ; i++) {
        if (strcmp (cmds[i], argv[ 1 ]) == 0) {
        /* llama la funcion correspondiente */
            return (*f[ i ])(client, interp, argc-1,
                    &argv[ 1 ]) ;
        }
    }
    /* comando no encontrado */
    sprintf (msg,"No existe el comando %s", argv[ 1 ]);
    Tcl_AppendResult (interp, msg, (char *) 0) ;
    return  TCL_ERROR ;
}
```

El funcionamiento es sencillo: busca el comando por su nombre y, de encontrarlo, lo manda llamar con el mismo formato de encabezado que tiene la función.

`Tcl_alg_AbreBD` es un ejemplo de una función que implementa uno de los nuevos comandos, tiene el esquema general que se muestra a continuación:

```
static int Tcl_alg_AbreBD (ClientData client,
                    Tcl_Interp *interp, int argc, char **argv)
{

    if (argc != 2) {
      Tcl_AppendResult (interp, OPEN_USAGE, (char *) 0) ;
      return TCL_ERROR ;
    }

    if (db_open) {        /* Ya se esta usando una base */
      Tcl_AppendResult (interp, "Hay una BD en uso", (char *) 0);
      return TCL_ERROR ;
    }

    /* Cuerpo de la función */

    strcpy (db_name, argv[ 1 ]) ;
    db_open = TRUE ;
    return TCL_OK ;
}
```

El encabezado es básicamente el mismo en todas las funciones que corresponden a los nuevos comandos y todo el manejo de los parámetros del comando se realiza a través de los parámetros de la función. El procedimiento genérico es comprobar si el comando fue llamado con el número correcto de parámetros; después, se puede

verificar alguna precondición. El cuerpo de la función puede ser tan grande y complejo como se requiera y ya sea en éste o al final de la función va una parte que regresa el valor de la función (con la instrucción `return`).

Cada comando de Tcl regresa un valor, que depende de cómo se ejecutó la función, y una cadena con el resultado de la misma. Para poder establecer cuál va a ser el resultado de la función, es preciso llenar la estructura `interp`, la cual tiene un campo llamado `interp-->result` el cual puede ser accesado con las funciones que para este objetivo proporciona Tcl, que son: `Tcl_AppendResult` la cual toma varias cadenas y las concatena dentro de la variable `interp-->result` y concluye la concatenación con un caráter nulo; `Tcl_AppendElement`, que toma la cadena que ya tiene la variable y le añade al final la cadena que toma como parámetro; `Tcl_ResetResult`, asigna el valor de cadena nula a la cadena de resultado.

La biblioteca de Tcl posee también un conjunto de funciones que permiten evaluar *scripts* o comandos propios de Tcl, como son: `Tcl_Eval`, que permite evaluar un comando de Tcl; si es compuesto por varias cadenas se puede usar `Tcl_VarEval`; si se tiene un archivo con estos comandos se puede usar la función `Tcl_EvalFile`.

Un ejemplo del uso de `Tcl_VarEval`, en donde se observa que el comando a ejecutar puede ser uno de los que se acaban de crear, verifica si existe una relación en la base de datos y regresa un 0 si no existe:

```
/* comprueba que la relacion exista */
existe = Tcl_VarEval (interp, "alg esrel ", argv[ 1 ],
                      (char *) 0);

if (strcmp (interp->result, "0") == 0) {
    Tcl_ResetResult (interp) ;
    sprintf (mesg, "Relacion:  %s, inexistente", argv[ 1 ]) ;
    Tcl_AppendResult (interp, mesg, (char *) 0) ;
    return TCL_ERROR ;
}
..
..
..
```

No se describirán más funciones de la biblioteca de Tcl, pero es importante mencionar en forma general que existen para el manejo de las variables de Tcl, para listas, tablas hash, cadenas y procesos.

4.3 Del álgebra relacional a Tcl

De acuerdo con lo mencionado en la sección anterior, se creó una extensión de Tcl llamada *Alg*, según el esquema de la página 50. Tenemos entonces el conjunto de comandos de Tcl complementado con aquellos que empiezan con `alg`, mismos que se encargan de interactuar con un manejador de bases de datos a fin de efectuar una tarea dada. Las funciones que son parte de la extensión son quienes se encargan de trasladar un enunciado en el lenguaje que estamos proponiendo como `alg` al lenguaje de manipulación de datos del manejador.

Del álgebra relacional naturalmente nos interesa ver la implementación de las funciones correspondientes a los operadores primarios que se analizaron en el capítulo 1. Tratando de apegarnos lo más posible a la forma sintáctica de los operadores del ágebra relacional presentada en el capítulo 1, pero manteniendo el estilo del lenguaje Tcl, se tomó la decisión de que los comandos correspondientes tuvieran el formato siguiente:

Unión: $R \times R \longmapsto R$	`alg union r s rs`
Inter: $R \times R \longmapsto R$	`alg inter r s rs`
Dif: $R \times R \longmapsto R$	`alg dif r s rs`
Cruz: $R \times R \longmapsto R$	`alg cruz r s rs`
Rest: $R \times$ *Condición* $\longmapsto R$	`alg rest r condicion rs`
Proy: $R \times$ *Atributos* $\longmapsto R$	`alg proy r atributos rs`
θ*Unión*: $R \times R \times$ *Condición* $\longmapsto R$	`alg theta r s condicion rs`
Div: $R \times R \longmapsto R$	`alg div r s rs`

Cada uno de los comandos de `alg` regresa como resultado el nombre de la relación resultado (el cual pasamos como último parámetro) o una cadena vacia para indicar errores. A manera de ejemplo se muestra la implementación de uno de ellos, la proyección:

```
/* Proyeccion <Rel_R> <atrib1> [<atribn>]* <Rel_RS> */
static int Tcl_alg_Proyeccion (ClientData client,
                    Tcl_Interp interp, int argc, char **argv)
{
    char  mesg[ 2048 ] ;
    int  existe , i ;
/* Chequeo de parámetros y precondiciones */
    ..
```

```
    ..
/* Ahora verifico que la relación fuente exista */
  existe = Tcl_VarEval (interp, "alg esrel ", argv[ 1 ],
              (char *) 0) ;
  if (strcmp (interp->result, "0") == 0) {
    Tcl_ResetResult (interp) ;
    sprintf (mesg,"Relacion:  %s, inexistente", argv[ 1 ]);
    Tcl_AppendResult (interp, mesg, ( char * ) 0) ;
    return  TCL_ERROR ;
  }

/* Se verifica que los atributos existan en la relacion */
  for (i = 2 ; i < argc-1 ; i++) {
    existe = Tcl_VarEval (interp, "alg esatrib ", argv[ 1 ],
              " ", argv[ i ], ( char * ) 0) ;
    if ( strcmp( interp->result, "0" ) == 0 ) {
      Tcl_ResetResult (interp) ;
      sprintf (mesg, "Atributo:  %s, no existe en %s",
              argv[ i ], argv[ 1 ]) ;
      Tcl_AppendResult (interp, mesg, ( char * ) 0) ;
      return TCL_ERROR ;
    }
  }

  Tcl_ResetResult (interp) ;

/* Ahora checo que no exista la relacion resultado */
  existe = Tcl_VarEval (interp, "alg esrel ", argv[ argc-1 ],
              (char *) 0) ;
  if (strcmp (interp->result, "1") == 0) {
    Tcl_ResetResult (interp) ;
    sprintf (mesg, "Relacion:  %s, ya existe.", argv[ 1 ]) ;
    Tcl_AppendResult (interp, mesg, (char *) 0) ;
    return  TCL_ERROR ;
  }
    ..
```

En esta parte se ve el uso de un par de comandos que complementan a la extensión y que hacen trabajo de más bajo nivel que los operadores mismos; uno verifica si existe o no una relación dentro de la base de datos y el otro comprueba si un nombre corresponde con alguno de los atributos de la relación. Estos son **alg esrel** <relacion>, y alg esatrib rel <atrib>.

El código siguiente corresponde a la interacción con el manejador, en este primer caso el manejador Ingres.

```
/* Construye el predicado del retrieve */
Tcl_ResetResult (interp) ;
IIwrite ("RANGE OF e IS ") ;
IIwrite (argv[ 1 ]) ;
IIsync (0) ;
IIwrite (mesg) ;

mesg[ 0 ] = '\0' ;
IIwrite ("RETRIEVE INTO ") ;
IIwrite (argv[ argc-1 ]) ;
IIwrite (" ( e.") ;
for (i = 2 ; i < argc-2 ; i++) {
  IIwrite (argv[ i ]) ;
  IIwrite (", e.") ;
}
IIwrite (argv[ argc-2 ]) ;
IIwrite (" ) ") ;
IIsync (0) ;

/* La función regresa el nombre de la relación resultado */
  Tcl_AppendResult (interp, argv[ argc-1 ], (char *) 0) ;
  return  TCL_OK ;
}
```

En este caso se utiliza el conjunto de funciones que ofrece en la biblioteca `libq`, la cual pertece a Ingres y proporciona las primitivas de comunicación con el manejador. Se supone que la base de datos ya esta abierta (lo cual se verificó en las precondiciones) y básicamente la interfaz se hace con la función `IIwrite`, la cual *alimenta* al manejador de su lenguaje de manipulación de datos. A continuación, se presenta el equivalente para mSQL:

```
/* Me conecto al manejador */
if ((lsock = msqlConnect (host)) < 0)  {
  Tcl_AppendResult (interp, msqlErrMsg, (char *) 0) ;
  return  TCL_ERROR ;
}
/* Selecciono la base de datos */
if (msqlSelectDB (lsock, db_name)  < 0) {
  Tcl_AppendResult (interp, msqlErrMsg, (char *) 0) ;
  return  TCL_ERROR ;
}
```

```
/* Construye la consulta */
strcat (query, "SELECT ") ;
for (i = 2 ; i < argc - 1 ; i++, nt++) {
  strcat (query, " ") ;
  strcat (query, argv[ i ]) ;
  strcat (query, ",") ;
  strcat (atributos, argv[ i ]) ;
  strcat (atributos, " ") ;
  code = Tcl_VarEval (interp, "alg tipo ", argv[1], " ",
                      argv[i], (char *) 0) ;
  strcat (atributos, interp->result) ;
  strcat (atributos, " ") ;
}

Tcl_ResetResult (interp) ;
```

mSQL trabaja bajo un modelo cliente–servidor sobre una red, de ahí que se tiene que decir cuál es la máquina que proporciona el servicio para poder conectarse mediante un *socket*. mSQL proporciona un subconjunto de SQL y uno de los problemas a resolver es que no es posible hacer una consulta y que el resultado de tal se baje a una nueva relación; es necesario hacer la consulta, crear una nueva relación y guardar en tal relación los resultados de la consulta.

```
/* Crea la relacion resultado */
if ((code = Tcl_VarEval (interp, "alg crearel ", argv[argc-1],
                  " ", atributos, (char *) 0) ) != TCL_OK) {
  return TCL_ERROR ;
}

n = strlen (query) ;
query[n-1] = '_ 0' ;
strcat (query, " FROM ") ;
strcat (query, argv[ 1 ]) ;

/* Ejecuta la consulta */
if ( msqlQuery (lsock, query) < 0) {
  msqlClose  (lsock) ;
  Tcl_AppendResult (interp, msqlErrMsg, (char *) 0) ;
  return TCL_ERROR  ;
}

Tcl_ResetResult (interp) ;
resultado = msqlStoreResult () ;
```

```
while ( (col = msqlFetchRow (resultado)) ) {
  sprintf (cadena, "alg inserta %s ", argv[argc-1]) ;
  msqlFieldSeek ( resultado, 0 ) ;
  for (i = 0; i < msqlNumFields (resultado) ; i++) {
    if (!( campo = msqlFetchField (resultado))) {
      Tcl_AppendResult (interp, msqlErrMsg, (char *) 0) ;
      msqlClose (lsock) ;
      return TCL_ERROR ;
    }
    if (col[ i ]) {
      if (campo->type == CHAR_TYPE) {
        tmp = escapeText (col[ i ]);
        sprintf (cadena,"%s %s '%s'", cadena, campo->name, tmp) ;
        free (tmp) ;
      } else {
        sprintf (cadena,"%s %s %s", cadena, campo->name,
                 col[ i ]) ;
      }
    }
  }

  code = Tcl_VarEval (interp, cadena) ;
  if ( code == TCL_ERROR ) {
    msqlClose (lsock) ;
    return  TCL_ERROR ;
  }
}

msqlFreeResult (resultado) ;
msqlClose (lsock) ;

Tcl_ResetResult (interp) ;
Tcl_AppendResult (interp, argv[ argc-1 ], (char *) 0) ;
return  TCL_OK ;
```

La idea en ambos manejadores es construir con los argumentos (los cuales se validan) el predicado correspondiente a la operación que deseamos implementar en el lenguaje de manipulación de datos correspondiente. Para cada operador, de realizarse satisfactoriamente la operación, regresa el nombre de la relación generada y de esta forma es posible hacer la composición de los operadores.

4.4 Comandos complementarios

Para poder realizar una aplicación en la que se utilicen los comandos de ALG correspondientes a los operadores del ágebra relacional, es necesario proporcionar un conjunto de comandos adicionales (que también son parte de la extensión) para poder realizar operaciones primitivas sobre una base de datos, como son: crear una base de datos, borrar una base de datos, listar las bases de datos existentes, etcetera. He aquí una tabla con la descripción de los comandos que complementan a *alg* como una extensión de Tcl:

Sobre Bases de Datos

alg abrebd bd
Abre una base de datos

alg cierrabd bd
Cerrar la base de datos

alg creabd bd
Crea una base de datos

alg borrabd bd
Borrar una base de datos

alg bases
Genera un lista de bases existentes

Sobre relaciones

alg rels
Genera una lista de relaciones de la base de datos en uso

alg crearel n_rel at_1 tipo_1 [at_i tipo_i]*
Crear una relación

alg borrarel n_rel
Borrar una relación

alg esrel n_rel
Función booleana para verificar si existe una relación

alg inserta n_rel at_1 val_1 [at_n val_n]*
Insertar una n-ada

alg borra n_rel at_1 val_1 [at_n val_n]*
Borrar una n-ada

`alg riguales n_rel1 n_rel2`
Función booleana que determina si dos relaciones son iguales

Sobre atributos

`alg atribs rel`
Genera una lista de atributos de una relación

`alg rena rel at_a at_b`
Renombrar un atributo

`alg tipo rel atrib`
Cual es el tipo de un atributo

`alg esatrib rel atrib`
Función booleana que determina si existe un atributo dentro de una relación

`alg tiposr rel`
Genera una lista de parejas atributo-tipo

Capítulo 5

Alg–Tk

Alg–Tk es un programa prototipo el cual proporciona una interfaz gráfica desarrollada en Tk y se basa en la extensión Alg de Tcl para emplear las funciones del manejo de las bases de datos y los operadores del álgebra relacional. La idea de este programa es tener una ventana que sea el panel de control principal en cual contiene un conjunto de menús para las operaciones básicas que son: manejo de las bases de datos, manejo de las relaciones, operaciones sobre n–adas y los operadores del álgebra relacional principalmente.

La forma de trabajar con este programa es la de seleccionar una base de datos para poder tarbajar con las relaciones, despues definir con que relación se va a trabajar para activar las operaciones sobre las n–adas; puede uno trabajar los operadores del álgebra relacional desde el momento en que se selecciona la base de datos. En las siguientes secciones se muestran las partes más significativas de este programa, el cual tiene en general la siguiente presentación.

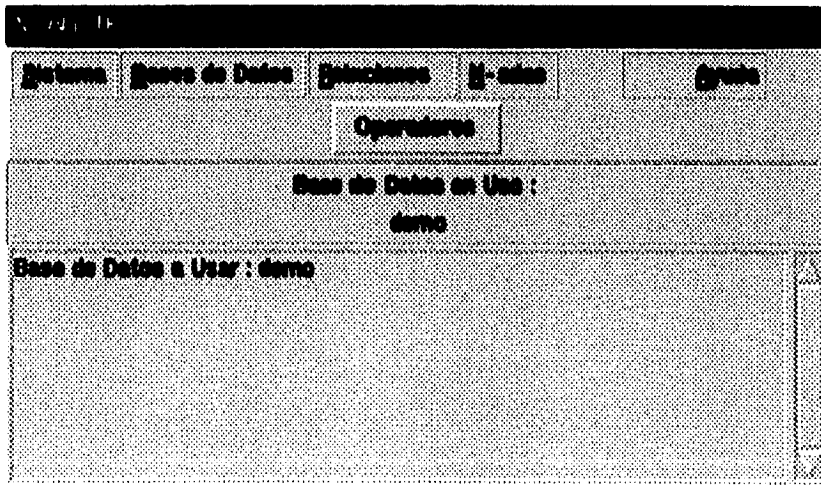


Figura 5.1 Vista general de Alg–Tk

5.1 Operaciones sobre las bases de datos

Las operaciones que se implementaron en el nivel bases de datos son: crear una base de datos (sin la definición de relaciones, la cual se encuentra en otra ventana), borrar bases de datos, listar o inspeccionar una base de datos (la inspección se realiza a nivel de las relaciones y atributos con sus tipos que se encuentran en cada base de datos) y definir que base de datos se va a usar en las operaciones siguientes.

5.1.1 Crear bases de datos

La ventana de crear relaciones básicamente se compone de una ventana tipo *toplevel* que contiene a su vez una ventana tipo *message* que muestra el mensaje de creación de una base de datos, una tipo *entry* que captura el nombre de la base de datos a crear, y un par de botones para realizar la operación o cancelar. Este ventana tiene la siguiente vista.

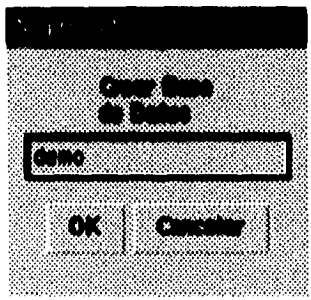


Figura 5.2 Crear una base de datos

5.1.2 Borrar bases de datos

Esta opción permite borrar bases de datos, las cuales se seleccionan a través de una lista que contiene los nombre de las bases de datos. Una vez seleccionada una base de datos, es posible inspeccionar el conjunto de relaciones que contiene para asegurarse que es la base de datos que se desea borrar y la operación se realiza finalmente cuando se presiona el botón de borrar o bien se puede cancelar la operación como se ve en la siguiente figura.

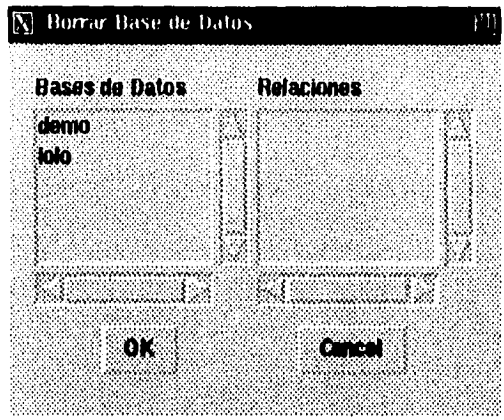


Figura 5.3 Borrar una base de datos

5.1.3 Listar bases de datos

Esta ventana permite inspeccionar cualquier base de datos en términos de las relaciones que contiene, los atributos de cada relación y sus correspondientes tipos. Esta formada por cuatro ventanas, cada una con un *listbox*, al seleccionar el nombre de una base de datos, se despliega la lista de relaciones y al escoger una relación se despliega la lista de atributos cada uno con su tipo correspondiente como se muestra en la siguiente figura.

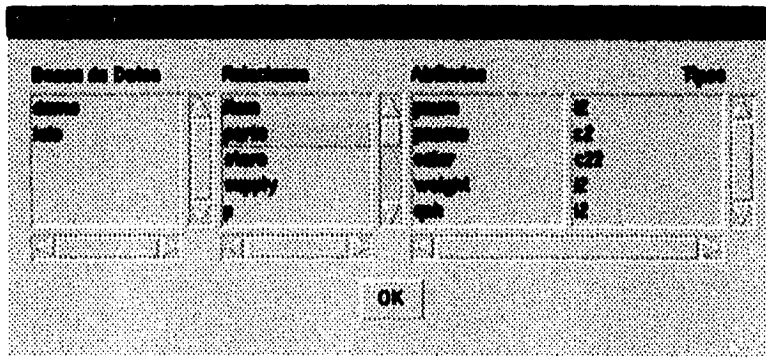


Figura 5.4 Listar bases de datos

5.1.4 Usar una base de datos

Para seleccionar una base de datos, la ventana difiere muy poco con respecto al de borrar las bases de datos porque aquí hay dos botónes uno de los cuales permite revisar que base de datos se desea y seleccionarla o bien se puede cancelar la operación.

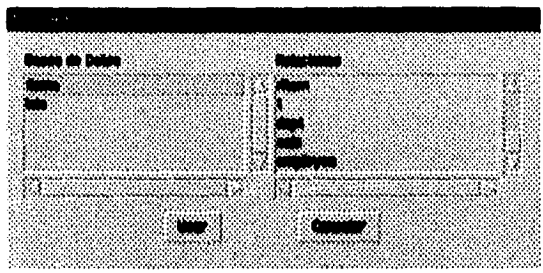


Figura 5.5 Usar una base de datos

5.2 Operaciones sobre las relaciones

Una vez que ha sido seleccionada la base de datos a usar, se activa la ventana para trabajar con las relaciones de tal base de datos. El conjunto de operaciones sobre las relaciones es: crear relaciones (se definen todos los atributos y tipos correspondientes), borrar relaciones (en donde se puede hacer una inspección sobre los atributos y tipos), listar las relaciones (básicamente una inspección de relaciones, atributos y tipos) y usar relación (en donde se define que relación se fija para la ventana de manejo de n-adas).

5.2.1 Crear relaciones

Esta opción consiste de una ventana con un listbox para la inspección de las relaciones ya existentes en la base de datos, un par de ventanas de tipo entry para capturar lo que será el nombre de la relación y de este último mediante la selección cada uno de los atributos. Una vez definido el nombre de la relación éste se fija y se inicia la definición de los atributos capturando el nombre y el tipo, en una lista.

Una vez que se definieron el nombre de la relación y el conjunto de atributos se tiene un botón para mandar ejecutar la creación de esta nueva relación en la base de datos y por otra parte cancelar la operación, la siguiente figura contiene la forma en que se muestra esta ventana.

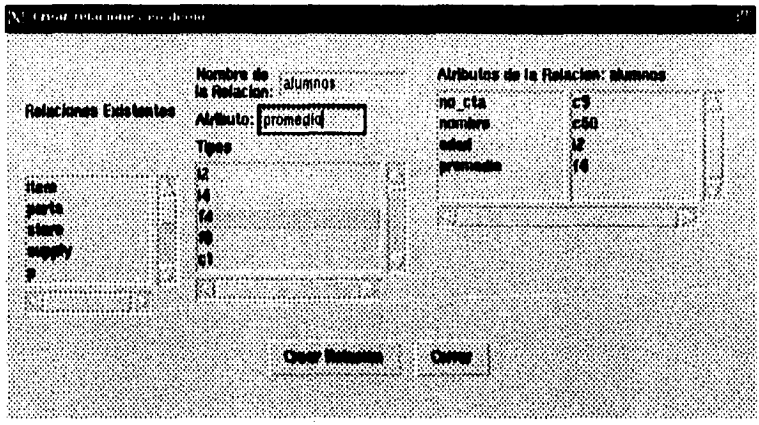


Figura 5.6 Crear relaciones

5.2.2 Borrar relaciones

Esta ventana permite inspeccionar las relaciones y sus atributos para borrar cualquier relación. La ventana consta de una lista de relaciones y una lista compuesta de parejas atributo–tipo, junto con un par de botónes para ejecutar el borrado de la relación o la cancelación de la operación. La forma de este ventana se muestra a continucación.

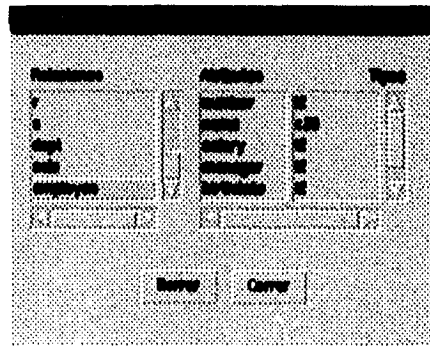


Figura 5.7 Borrar relaciones

5.2.3 Listar relaciones

La ventana que corresponde a esta opción consiste de una lista con las relaciones y una lista compuesta de atributo-tipo, la cual sirve para inspeccionar la forma en que están formadas las relaciones, la ventana se muestra a continuación.

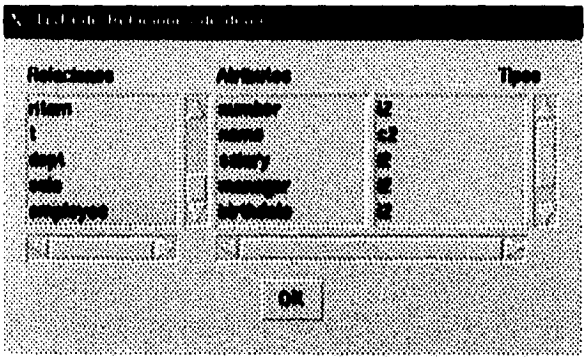


Figura 5.8 Listar relaciones

5.2.4 Usar relaciones

Esta opción es en esencia igual al de borrar relaciones. Cuando se hace la elección de una relación se activa el menú de operaciones sobre *n-adas* del menú principal. La ventana para seleccionar una relación se muestra a continuación.

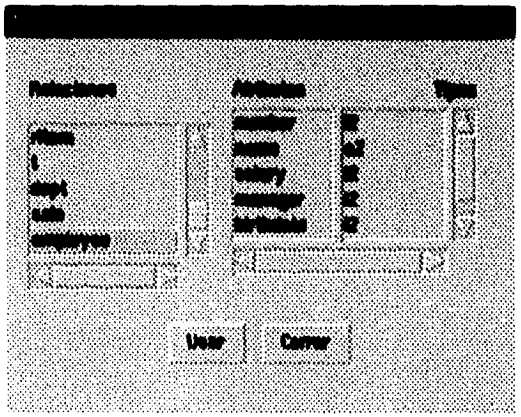


Figura 5.9 Usar relaciones

5.3 Operaciones sobre las n-adas

Las operaciones definidas sobre las n-adas son: insertar una n-ada sobre una relación preestablecida, borrar, modificar e inspeccionar las n-adas de la relación. En estas opciones se utilizaron la ventana de insección de una n-ada la cual se reutiliza en la opción de modificar n-ada ya que es la misma operación pero con un conjunto de valores predeterminados. De igual manera la ventana de inspeccionar n-adas es la misma que en el borrado pero con la capacidad de marcar un conjunto de n-adas a ser borradas, o en la modificación de n-adas en la cual se selecciona una a la vez.

5.3.1 Inspeccionar n-adas

Se desarrolló una interfaz en la cual fuese posible presentar un conjunto de listas que contienen atributos, y donde cada n-ada esta formada por los elementos al mismo nivel en cada una de las listas. Este manejo permite que durante la inspección cada una de las listas se mantengan exactamente al mismo nivel.

La parte alta de esta ventana muestra los nombres de cada uno de los atributos y debido a que todo el conjunto de atributos no cabe necesariamente en la ventana, se realiza un desplazamiento horizontal de las listas de nombres de los atributos. En caso de que algún valor dentro de un atributo fuese mas largo que el ancho de la lista es posible hacer un desplazamiento horizontal por lista a fin de visualizar completamente los valores de los atributos. La ventana tiene la siguiente forma.

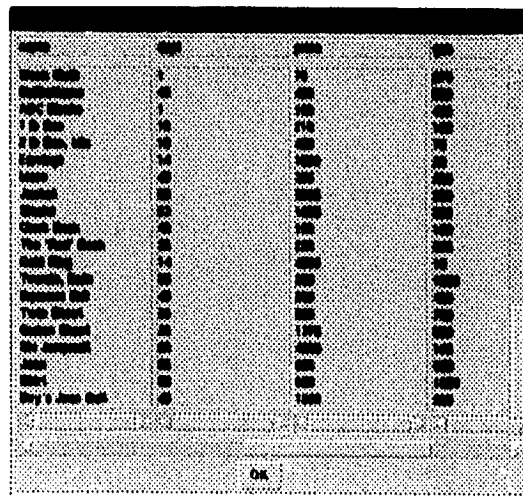


Figura 5.10 Inspeccionar relaciones

5.3.2 Insertar n-adas

La forma de insertar es a través de una lista de pares de etiquetas que tienen el nombre del atributo y su tipo; junto con un ventana tipo entry para capturar el valor del atributo, una vez que se ha completado toda la n-ada se captura por medio del botón de insertar y existe otro para cancelar la operación o bien para dejar de capturar n-adas.

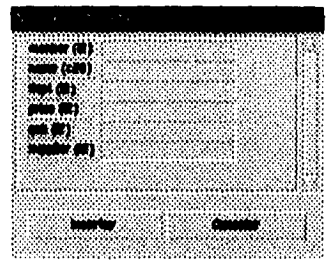


Figura 5.11 Insertar n-adas

5.3.3 Borrar n-adas

Para borrar n-adas se tiene un interfaz en la cual podemos inspeccionar toda la relación he ir seleccionando una a la vez y borrando o marcar todo un conjunto y aplicar la operación de borrar sobre estas. De manera natural esta ventana es la misma que la de inspeccionar n-adas, pero con la característica de poder marcar n-adas y ejecutar una operación sobre ellas (la de borrar).

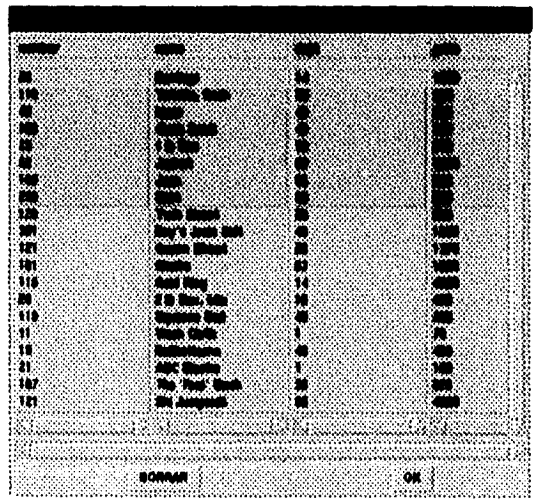


Figura 5.12 Borrar n-adas

5.3.4 Modificar n-adas

Para modificar n-adas se creo una interfaz como la de inspeccionar n-adas, a partir de la cual se hace una selección por cada una de las n-adas a modificar pasando entonces a una ventana como la de insertar n-adas pero con los valores ya predefinidos de la n-ada seleccionada.

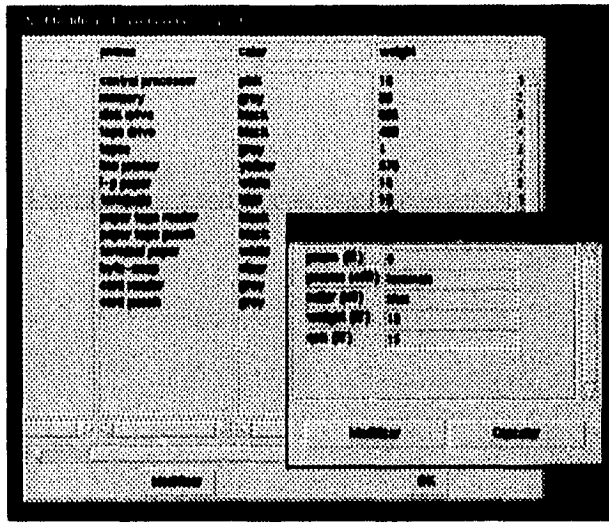


Figura 5.13 Modificar relaciones

5.4 La interfaz de los operadores

Una vez que se ha seleccionado una base de datos es posible trabajar con los operadores del ágebra relacional. Al nivel del menu principal se tiene un boton que activa una ventana de tipo *toplevel* donde se tiene la interfaz a los ocho operadores del álgebra relacional.

Esta ventana contiene una lista de botones, uno para cada operador, y al seleccionar uno de ellos aparece una interfaz particular en la parte baja de la ventana, como se muestra en la figura 5.14. De acuerdo a la definición de cada uno de los operadores se tiene en general un boton con el nombre del operador; un par de botones que equivalen al par de relaciones r y s, cada uno de los cuales despliega una lista de las relaciones existentes en la base de datos; un boton mas que al seleccionarlo permite capturar el nombre de la relación resultado.

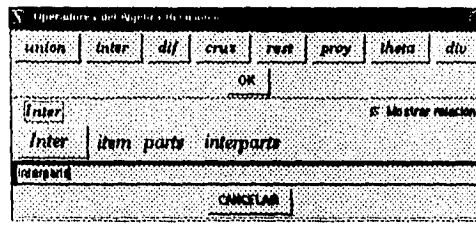


Figura 5.14 Operadores del álgebra relacional

Una vez que se han seleccionado las relaciones y atributos (en los operadores donde aplica) junto con el nombre de la relación resultado, esto activa el boton del operador para mandar ejecutar la operación, la cual, en caso de que el *checkbutton* de mostrar la relación resultado este activado entonces procederá a mostrar la relación en una ventana.

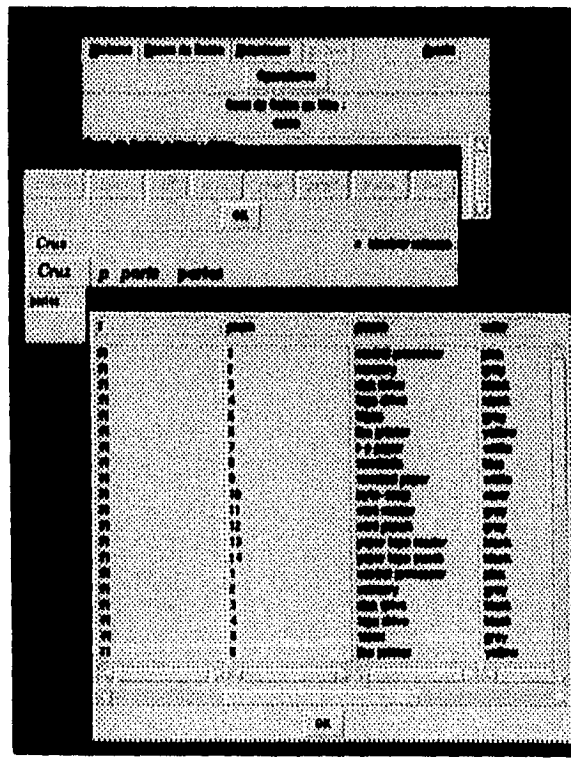


Figura 5.15 Vista general de los operadores

Cinco operadores (unión, intersección, diferencia, producto cruz y división) se manejan bajo este mismo esquema. En la restricción y la proyección en lugar del segundo boton de relación se encuentra una lista con los atributos sobre los cuales se realizará la operación, como se muestra en la figura 5.16; finalmente en el caso de la theta-unión se tienen ambas relaciones junto con sus atributos a seleccionar (i.e. cuatro botones de listas, dos que muestran relaciones y dos que muestran atributos).

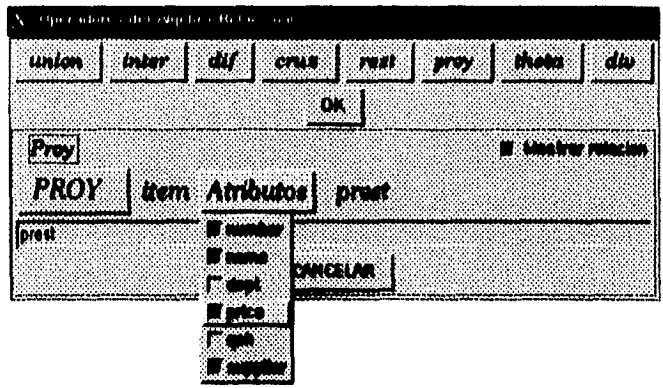


Figura 5.16 Operador proyección

Al presionar el botón del operador se ejecuta el comando asociado a éste; por ejemplo en el caso del operador proyección ejecuta el siguiente código:

```
.opera.f.forma configure -command {
    alg abrebd $dbname
# ejecuta el operador
    if [catch {alg proy $relar $ratribs $rel} mesg] {
        alg cierrabd
        Dialogo .f {ERROR} $mesg {} -1 Continuar
    } else {
        alg cierrabd
# añade la nueva relación a la lista
        .opera.f.mb1.menu add radio -label $rel \
            -variable relar -value $rel \
            -command ".opera.f.mb1 configure -text $rel"
# inspecciona la nueva relación
        if { $mostrar == 1 } {
            set relname $rel
            Inspec
        }
    }
}
```

Conclusiones

La aparición de Tcl ha sido sin duda un avance en los lenguajes de tipo *script*; éste, junto con su toolkit Tk, han tenido enorme impacto por las ventajas que han proporcionado para el diseño de interfaces gráficas sobre ambientes como X–Window, Windows o Macintosh, reduciendo enormemente la complejidad en el desarrollo de las aplicaciones. Otra característica importante ha sido la forma clara en que se pueden desarrollar extensiones al conjunto de comandos a través de toda una colección de herramientas que proporciona Tcl.

En general, ahora podemos observar no solo a Tcl sino a otros lenguajes como *Perl*, *Python* y otros, con la posibilidad de extender los conjuntos básicos de comandos, como una manera de que el lenguaje se acerque más al problema que se necesita resolver y que se ofrezca un mayor número de herramientas para reducir la complejidad de la solución.

Esta última capacidad ha permitido acercar conceptos abstractos como son los del álgebra relacional a Tcl, de forma que se utilicen comandos que emulan prácticamente el mismo nivel abstracto los operadores del ágebra relacional dentro de un programa.

El poder utilizar una herramienta como Alg permite entender los conceptos básicos del ágebra relacional que son fundamentales para el modelo relacional, el cual, a pesar de sus más de 25 años, sigue siendo la base para el desarrollo de la mayoría de los sistemas de información. Esta herramienta además proporciona una independencia del manejador de bases de datos y de su lenguaje de manipulación de datos, en el cual no siempre podemos ubicar estos conceptos básicos del ágebra relacional. Además, siendo Alg una extensión, permite aprovechar todas las características de Tcl como un lenguaje que permite integrar y desarrollar sistemas.

Alg–Tk es un programa prototipo de una interfaz gráfica para un manejador de bases de datos, el cual ofrece trabajar al nivel de los conceptos fundamentales del álgebra relacional. La interfaz gráfica tiene sin duda grandes ventajas para el aprendizaje de los conceptos del modelo relacional, y quedan abiertas muchas posibilidades sobre posibles mejoras, opciones y formas de comunicarse con el usuario final. El hecho de que este programa haya sido desarrollado en Tcl/Tk ofrece la posibilidad de ser transportado a otras plataformas de desarrollo, mientras que la extensión Alg puede ser adaptada fácilmente para trabajar con el manejador de bases de datos que se tenga al alcance y sin modificar una sola línea de Alg–Tk.

Alg–Tk tambien puede verse como la interfaz gráfica con la que no cuentan tanto mSQL como Ingres, así que puede ser modificada fácilmente para explotar las características particulares de cada uno de ellos.

El trabajo realizado ha permitido ententer por un lado la filosofía del lenguaje Tcl y su extensión Tk; entender dos manejadores de bases de datos Ingres y mSQL, con origenes, lenguajes y formas totalmente ajenas de trabajar para ofrecer mediante la extensión Alg, una forma de trabajo independiente a estos.

El trabajo a futuro puede ir en varios sentidos. Uno de ellos es extender aun más las posibilidades de Alg–Tk con un generador de reportes, un generador de pantallas de captura y otras más que hagan de esta interfaz todo un ambiente de trabajo relacionado con las bases de datos. En la parte de Alg como extensión que implementa el álgebra relacional faltaría ver las posibilidades respecto a completar en un 100% al modelo relacional, esto con respecto al manejo de las reglas de integridad, llaves primarias, llaves foráneas y el manejo de éstas dentro de los operadores del álgebra relacional.

Bibliografía

[ALLM76] Allman E, M. Stonebraker and G. Held. *Embedding a relational data sublanguage in a general purpose programming language.* ACM-SIGMOD International Conference on Management of Data, Salt Lake City, Utah, March 1976.

[CODD70] Cood, E.F. *A relational model for large shared data banks.* Communications of ACM 13, 6(June 1970), pp. 377-387.

[CODD71] Codd, E.F. *A data base sublanguage founded on relational calculus.* ASM SIGFUDET Workshop on Data Description, Access and Control, 1971, pp. 35-68.

[CODD81] Codd, E.F. *Data models in database management.* ACM-SIGMOD Record 11, No. 2(February 1981).

[CODD82] Codd, E.F. *Relational database: A practical foundation for productivity.* Communications of the ACM, February 1982, Vol. 25, No. 2.

[EPST77] Epstein, Robert. *Creating and maintaining a database using INGRES.* Memorandum No. ERL-M-77-77, December 16, 1977. Electronics Research Laboratory, College of Engineering, University of California, Berkeley.

[DATE91] Date, C. J. *An introduction to Database Systems.* Volume I, Fifth Edition, Addison Wesley 1991.

[HAWT79] Hawthorn P., M. Stonebraker. *The use of Technological Advances to Enhance Database System Performance.* ACM-SIGMOD Conference Proceedings International Conference on Management of Data, Boston, Massachusetts, June 1979.

[HUGHES] Hughes, David J. *Mini SQL. A Lightwight Database Engine.* ftp://ftp.bond.edu/pub/Minerva/msql/msql-1.0.16.tar.gz

[IBAR94] Ibargüengoitia Glez. Guadalupe E. *Principios Matemáticos de las Bases de Datos Relacionales.* Cuadernos del seminario Temas de Cómputo, Vol. I, Número 1, 1994. Facultad de Ciencias, UNAM.

[ICAZ94] De Icaza Amozurrieta, Miguel. *El Lenguaje Tcl.* Revista Soluciones Avanzadas No. 11, Julio 1994.

[INGR93] Ingres. Version 8 Reference Manual, 4/16/93. Joe Kalash, Lisa Rodgin, Zelaine Fong, Jeff Anton.

[KERN70] Kernighan, Ritchie. *El Lenguaje de Programación C.* Prentice Hall, 1970.

[KNUTH] Knuth, Donald Ervin. The TEXbook. Addison Wesley, 1990.

[OUST90] Ousterhout, John K. *Tcl: An Embeddable Command Language.* USENIX Proceedings 1990.

[OUST94] Ousterhout, John K. *Tcl and the Tk Toolkit.* Addison Wesley, professional computing series, 1994.

[OUST96] Ousterhout, John K. *An Introduction to Tcl and Tk.* USENIX 1996 Annual Technical Conference.

[STON76] Stonebraker M., Peter Kreps, Eugene Wong, Gerald Held. *The Design and Implementation of INGRES.* ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976.

[STON80] Stonebraker, Michael. *Retrospection on a Database System.* ACM Transactions on Database Systems, vol. 5, No. 2, june 1980.

[STON81] Stonebraker, Michael. *Operating System Support for Database Management.* Communications of the ACM, vol. 24, no. 7, July 1981.

[STON83] Stonebraker, M., J. Woodfill, J. Ranstrom, M. Murphy, M. Meyer, E. Allman. *Performance Enhancements to a Relational Database System* ACM Transactions on Database Systems, vol. 8, no. 2, june 1983.

[RIES79] Ries Daniel R., M. Stonebraker. *Locking Granularity Revisited.* ACM Transactions on Database Systems, vol. 4, no. 2, June 1979.

[RIVE88] Rivero Cornelio E. *Bases de Datos Relacionales.* Parainfo 1988.

[ROWE90] Rowe, Lawrence A., M. Stonebraker. *The Commercial INGRES Epilogue.* en C. J. Date, *Relational Database Writings 1985-1989.* Addison Wesley 1990.

[SAHA93] Sah Adam, J. Blow. *A Compiler for the Tcl Language.* USENIX Proceedings 1993.

[UHLER96] Uhler Steve, Welch Brent. *Advanced Programming with Tcl and Tk.* Tutorial materials, USENIX 1996 Annual Techinical Conference, 1996.

[WELC95] Welch Brent.*Practical Programming in Tcl and Tk.* Prentice Hall, 1995.

[WONK79] Won Kim. *Relational Database Systems.* Computing Surveys, vol. 11, No. 3, September 1979.

[YOUS79] Youssefi Karel, E. Wong. *Query Processing in a Relational Database Management System.* Proceeding of the Fifth Very large Data Base Conference, Rio de Janeiro 1979.