37
291

Universidad Nacional Autónoma de México

FACULTAD DE CIENCIAS

UN MANEJADOR DE BASE DE DATOS RELACIONAL IMPLEMENTADO EN PROLOG

T E S I S

Que para obtener el título de

M A T E M A T I C O

p r e s e n t a

ADALBERTO VELAZQUEZ MENDEZ

Director de Tesis: Mat. Guadalupe E. Ibargüengoitia G.

México, D.F. 1991

FALLA DE ORIGEN

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

TESIS CON FALLA DE ORIGEN

D I R E C T O R

MAT. MA. GUADALUPE ELENA IBARGUENGOITIA GONZALES

S I N O D A L E S

MAT. FACUNDO RUIZ DONCEL

MAT. JAVIER SANCHEZ FLORES

M. EN C. AMPARO LOPEZ GAONA

ACT. GUSTAVO MARQUEZ FLORES

UN MANEJADOR DE BASE DE DATOS RELACIONAL IMPLEMENTADO EN PROLOG

INTRODUCCION.

CAPITULO 1 : PROGRAMACION LOGICA.

1.1 Concepto de programación lógica.
1.2 Lógica de predicados de primer orden.
1.3 Cláusulas de Horn.
1.4 Consecuencia Lógica.
1.5 Reglas de Inferencia.
1.6 Interpretación procedural.

CAPITULO 2 : PROLOG.

2.1 PROLOG como un lenguaje de Programación Lógica.
2.2 Estructuras de datos.
2.3 Estructuras declarativas.
2.4 Comportamiento computacional.
2.5 Características extralógicas.

CAPITULO 3 : BASES DE DATOS RELACIONALES.

3.1 Bases de Datos y sus características.
3.2 Conceptos básicos del Modelo de datos Relacional (Terminología).
3.3 Algebra Relacional.
3.4 Cálculo Relacional.
3.5 Relación con la Lógica de predicados de primer orden.

CAPITULO 4 : DESARROLLO DE PROGRAMAS EN PROLOG PARA LA DEFINICION, MANIPULACION E INTERROGACION DE BASES DE DATOS RELACIONALES.

4.1 Definición.
4.2 Manipulación.
4.3 Consulta.
4.4 Interfaz con el usuario
4.5 Ejemplo de operación.

CONCLUSIONES.

ANEXO A : PROGRAMA FUENTE.

ANEXO B : BIBLIOGRAFIA.

INTRODUCCION

El presente trabajo tiene como objetivo mostrar cómo las técnicas desarrolladas en el área de la Programación Lógica pueden ser utilizadas en la de las Bases de Datos para construir un Manejador de Base de Datos Relacional.

El interés en este tema surge al tener los primeros contactos con el lenguaje de programación Prolog.

En el Capítulo 1 se presenta la fundamentación lógica del lenguaje de programación basado en un subconjunto de la lógica de predicados.

Tomando como punto de partida la definición de una Cláusula y eligiendo un subconjunto de éstas, llamado Cláusulas de Horn, llegamos al conjunto que conforma el lenguaje de programación en cuestión.

Una vez que se ha definido cuándo un conjunto de estas cláusulas conforma un Programa Lógico, se pasa a la familiarización con los conceptos de Consecuencia Lógica y Reglas de Inferencia con los cuales se pueden definir nuevas cláusulas a partir de otras.

Por último se muestra a las cláusulas de Horn como un lenguaje de programación, estableciendo un paralelismo entre la forma de ejecución de un programa procedural y la ejecución de un programa lógico.

En el Cápitulo 2 se da una introducción al lenguaje de programación Prolog, mostrando las características que lo clasifican como un lenguaje de programación lógica basado en cláusulas de Horn.

El orden de ejecución de un programa en Prolog, que difiere de la forma en como se ejecuta un programa lógico es presentado, así como un grupo de características extralógicas que facilitan la construcción de programas agregando eficiencia.

En el Cápitulo 3 se describen las Bases de Datos Relacionales, se revisan sus conceptos básicos y se dan los requerimientos mínimos para que una aplicación pueda ser considerada como un Manejador de Base de Datos Relacional.

Se establece para finalizar la relación entre las bases de datos relacionales y la lógica de predicados de primer orden, dando pie al objetivo primordial de este trabajo.

En el capítulo 4 se desarrollan procedimientos en Prolog, que implementan el conjunto mínimo de operaciones que debe contemplar un manejador de base de datos relacional.

Además de estos procedimientos, se describe otros que permiten aceptar consultas a la base de datos, en forma similar al lenguaje de manipulación de datos de nombre SQL.

Los procedimientos mencionados, que conforman la aplicación total, se listan en el anexo-A.

CAPITULO 1

PROGRAMACION LOGICA

En este capitulo se centra la atención en la fundamentación lógica del lenguaje de programación basado en un subconjunto de la Lógica de predicados de primer orden.

1.1 CONCEPTO DE PROGRAMACION LOGICA.

Conocemos al menos tres estilos de lenguajes de programación: Procedural, Funcional y Relacional. Cada uno de ellos comprende un enfoque único de programación, existiendo diferencias tan marcadas que la idea que se tiene de un algoritmo puede ser fuertemente influenciada por el estilo del lenguaje de programación que se escoge.

En un lenguaje procedural o imperativo, (como Pascal, Fortran, C,...), se especifica una serie de pasos que describe la forma de resolver un problema, a esta serie de pasos se le conoce como un Programa Procedural y a cada uno de los pasos que la conforman como Instrucciones. Cuando un lenguaje procedural es usado para modelar un proceso en un medio ambiente, la relación entre el programa y el medio ambiente que está siendo modelado es de simulación.

En un lenguaje funcional (como Lisp, ML,..), se definen valores. La unidad atómica con la cual los programas funcionales son construidos es la **Función**. Podemos entender un programa funcional operacionalmente, evaluando funciones hasta que determinemos el valor final, o podemos entenderlo declarativamente, como una definición del valor deseado.

En un lenguaje relacional (Prolog o algún otro lenguaje de programación lógica), se especifica una relación entre individuos. Esta relación está definida en términos de condiciones o restricciones que, cuando son satisfechas, implican que la relación se cumple. Cuando un lenguaje de programación lógica es usado para modelar un proceso en un medio ambiente, la relación entre el programa y el medio ambiente que está siendo modelado es de descripción. El programa se encuentra en un nivel de abstracción mas alto que el proceso que describe, porque contiene afirmaciones y relaciones (abstracciones) acerca de él.

La solución en un lenguaje de programación lógica está muy cercana a la relación entre abuelos, padres e hijos. La descripción de ancestro establecería que :

"A es un ancestro de B si A es el padre de B o si A es un ancestro del padre de B"

Si representamos las relaciones:

A es un ancestro de B.
A es padre de B.

como,

ancestro (A,B)
padre (A,B)

y leyendo al símbolo ":-" como "si", nuestra solución quedaría escrita

ancestro (A,B) :- padre (A,B)
ancestro (A,B) :- padre (C,B) y ancestro (A,C)

Notemos que al mismo tiempo que se describe la relación entre personas y sus ancestros, también se define una forma de determinar si una persona es un ancestro de otra.

La programación lógica está basada en la idea de que un subconjunto de la lógica de predicados de primer orden puede ser utilizado como lenguaje de programación. Es por esto que a continuación daremos una breve descripción de esta forma de la lógica.

1.2 <u>LOGICA DE PREDICADOS DE PRIMER ORDEN</u>

En esta sección se muestra una definición de la Lógica de predicados de primer orden y la sintaxis de las fórmulas que la conforman, la cual será heredada al subconjunto de esta que es utilizado como lenguaje de programación.

Un Alfabeto de la Lógica de predicados de primer orden está formado por Términos, Predicados, Conectivos, Cuantificadores y Símbolos de puntuación; los cuales definiremos brevemente.

(a) Si p es un predicado n-ario y $t_1,..,t_n$ son términos, entonces $p(t_1,..,t_n)$ es una fórmula (llamada fórmula atómica o átomo).

(b) Si A y B son fórmulas, también lo son:

(A)
A , B
B <- A
A <-> B
A v B
not (A)

(c) Si F es una fórmula y x una variable, entonces tambien son fórmulas

$\exists x$ F $\forall x$ F

El Lenguaje de la Lógica de primer orden dado por un cierto alfabeto consiste en el conjunto de fórmulas bien formadas construidas a partir de símbolos de ese alfabeto. Este conjunto de fórmulas también es conocido como Lógica de Predicados de Primer Orden o Cálculo de Predicados.

Siguiendo estas definiciones podemos expresar enunciados a través de fórmulas de la Lógica de predicados. Ejemplos:

i) Oscar es un hombre.

hombre(Oscar)

ii) Todos los hombres son mortales.

(∀x)(hombre(x) -> mortal(x))

iii) Juan se sienta entre Ana y Luis.

entre(ana,juan,luis)

A continuación daremos unas definiciones útiles.

ALCANCE . La fórmula prefijada por un cuantificador es llamada su alcance; es decir, el alcance de ∀x en ∀x F es F. La ocurrencia de una variable está "acotada" si ocurre como variable de un cuantificador o si aparece en el alcance de un cuantificador que tiene como variable a ella misma. Toda variable que no es acotada es "libre".

FORMULA CERRADA O ENUNCIADO . Es una fórmula que no tiene ocurrencias libres de variables.

FORMULA ATOMICA . Es una fórmula que no contiene conectivos ni cuantificadores, es decir, es un predicado. Toda fórmula atómica o su negación son conocidos como "átomos" o "literales".

CLAUSULA . Es una disyunción de literales. Todas las cláusulas son prefijadas por cuantificadores universales, uno por cada variable que aparece en la cláusula, Así

$$\forall\ x_1,..,x_s\ (L_1\ v\ ..\ v\ L_m)$$

es una cláusula donde cada L_i es una literal y $x_1,..,x_s$ son todas las variables que ocurren en $(L_1\ v\ ..\ v\ L_m)$.

Dado que las L_i son literales (átomos o negaciones de átomos) podemos reescribir una cláusula de la forma siguiente:

$$\forall\ x_1,..,x_s\ (A_1\ v\ ..\ v\ A_k\ v\ not(B_1)\ v\ ..\ v\ not(B_n))$$

donde $A_1,..,A_k,B_1,..,B_n$ son átomos y $x_1,..,x_s$ las variables que ocurren en ellos.

Haciendo uso de algunas equivalencias de notación como son:

(i) not(A,B) = not(A) v not(B)
(ii) A -> B = not(A) v B

encontramos que, por (i), la fórmula anterior es equivalente a :

$$\forall\ x_1,..,x_s\ (A_1\ v\ ..\ v\ A_k\ v\ not(B_1,..,B_n))$$

la cual, usando (ii), a su vez es equivalente a :

$$\forall\ x_1,..,x_s\ (A_1\ v\ ..\ v\ A_k <- B_1,..,B_n)$$

Y así, asumiendo que todas las variables están universalmente cuantificadas, que las comas en el antecedente (como se conoce a $B_1,..,B_n$) denotan conjunción, y que las comas en el consecuente (como se conoce a $A_1,..,A_k$) denotan disyunción, obtenemos la notacion clausal :

$$A_1,..,A_k <- B_1,..,B_n$$

donde $B_1,..,B_n,A_1,..,A_k$ son literales, n,k >= 0.

Si la cláusula contiene las variables $x_1,..,x_m$ ésta se interpreta de la siguiente manera : Para toda $x_1,..,x_m$ A_1 o A_2 o .. o A_k se cumplen si B_1 y B_2 y .. y B_n se cumplen.

Será de suma importancia para nosotros un subconjunto de éstas formas clausulares llamado "Cláusulas de Horn", el cual definiremos a continuación.

1.3 CLAUSULAS DE HORN

Las cláusulas que contienen a lo más una literal en el consecuente, es decir, que tienen a lo más una literal no negada son llamadas Cláusulas de Horn. Estas tienen la forma :

$$A <- B_1,...,B_n$$

A es llamada la "cabeza" y B1,...,Bn es llamado el "cuerpo" de la cláusula.

Existen cuatro formas de cláusulas de Horn.

CLAUSULA DE PROGRAMA . Es una cláusula con exactamente una literal no negada .A estas cláusulas se les conoce como Reglas y Hechos:

$A <- B_1,...,B_n$: Reglas

A <- : Hechos

CLAUSULA META . Es una cláusula con ninguna literal no negada ,como son las Metas o un a cláusula especial llamada Cláusula vacía:

$<- B_1,...,B_n$: Metas

<- : **Cláusula Vacía**

PROGRAMA LOGICO . Es un conjunto finito de cláusulas de programa.

De las definiciones anteriores podemos observar que :

- Una cláusula de Horn es una cláusula de programa o una cláusula meta.

- Un hecho es una cláusula con un cuerpo vacío y una literal no negada, y su significado es: "El predicado A es verdadero".

- Una regla es una cláusula con una literal no negada y al menos una literal negada cuyo significado es: "Si $B_1,...,B_n$ son verdaderos entonces A es verdadero".

- Una meta es una cláusula con cabeza vacía y al menos una literal negada y se interpreta como una disyunción de literales negadas de la siguiente forma:

$$\forall x_1,...,x_n \ not(B_1) \ v \ ... \ v \ not(B_n)$$

la cual es lógicamente equivalente a:

$$\forall\ x_1,\ldots,x_n\ \text{not}(B_1,\ldots,B_n)$$

y también a:

$$\text{not}\ \exists\ x_1,\ldots,x_n\ (B_1,\ldots,B_n)$$

por lo tanto, su significado es: "No existen variables $x_1,\ldots,x_n$ tales que los predicados $B_1,\ldots,B_n$ sean verdaderos".

- Una cláusula vacía es la que no tiene cabeza y tampoco cuerpo, su significado es una contradicción con las cláusulas de programa y se la asigna el símbolo $\square$.

- Una cláusula de Horn es una implicación en la cual la conjunción de cero o más condiciones implica a lo más una conclusión.

Como notación convendremos que los nombres de variables empiezan con una letra mayúscula, y que los nombres que empiezan con letras minúsculas corresponden a constantes, predicados y funciones.

Basándonos en estas reglas de notación podemos escribir cláusulas de Horn como las siguientes:

```
1)    d(c,X,X) <-

2)    d(s(X),Y,s(Z)) <- d(X,Y,Z)

3)    vive_en(juan,mexico) <-

4)    paga_mucha_renta(X) <- vive_en(X,mexico)

5)    <- paga_mucha_renta(X)
```

Fórmulas de esta clase son enunciados puramente formales. No podemos saber si son verdaderos o falsos, porque no se ha asignado un significado particular a cada uno de los símbolos de predicados y de funciones.

Las cláusulas de Horn nos ofrecen una forma de representar conocimiento (afirmaciones y relaciones de un proceso determinado), veremos que también nos ofrecen una buena forma de razonar con ese conocimiento.

1.4 CONSECUENCIA LOGICA

Para poder hablar de verdad o falsedad de una fórmula es necesario asignar un significado a cada uno de los símbolos de la fórmula. Los cuantificadores y conectivos tienen un significado fijo, pero el significado asociado a las constantes, funciones y predicados puede variar.

Una Interpretación de un lenguaje L de primer orden consiste en:

(a) Un conjunto no-vacío D, llamado el "Dominio" de la interpretación.

(b) A cada constante en L, se le asigna un elemento en el dominio D.

(c) Para cada función n-aria en L, la asignación es un mapeo de D^n en D.

(d) Para cada predicado n-ario en L, la asignación de un mapeo de D^n en el conjunto {verdadero, falso}.

Una interpretación mapea una fórmula en un enunciado el cual es verdadero o falso, dependiendo del valor de verdad (es decir de la asignación de verdadero o falso) del enunciado.

Veamos los siguientes ejemplos:

1) Dada la fórmula d(z,X,X)

- Dominio N = Naturales
- z representa al 1
- d(X,Y,Z) es verdadero si y solo si XY = Z

La fórmula se convierte en el enunciado verdadero

"Para todo número natural X, 1X = X"

2) Dada la fórmula d(s(X),Y,s(Z)) <- d(X,Y,Z)

- Dominio N = Naturales
- d(X,Y,Z) es verdadero si y solo si XY = Z
- s representa a la función s(X) = 2X

La fórmula se convierte en el enunciado verdadero

"Para todos números naturales X,Y y Z, XY = Z implica 2XY = 2Z"

Si una interpretación mapea una fórmula en un enunciado verdadero entonces se llama un modelo de la fórmula. Podemos hablar también de un modelo de un conjunto de fórmulas, es decir, una interpretación en la cual todas ellas son verdaderas.

Existen fórmulas que son verdaderas para toda interpretación ,por ejemplo (A v not(A)), y también existen fórmulas que son falsas en toda interpretación como (A,not(A)) donde la "," denota conjunción. La primera clase de fórmulas son llamadas "*tautologías*", mientras que las segundas son llamadas "*inconsistencias*".

Estamos en este momento en posición de dar una definición concreta de consecuencia lógica :

Decimos que una fórmula s es consecuencia lógica de un conjunto de fórmulas S = $\{s_1, s_2, \ldots, s_n\}$, y lo denotamos S |= s, si todos los modelos del conjunto S lo son también de s.

Podemos notar que sólo se toma en cuenta la forma sintáctica de las fórmulas, sin importar sus interpretaciones. Esta definición es por lo tanto independiente de cualquier conexión que pueda ser establecida entre el lenguaje y el dominio de éste.

Sería una tarea muy difícil, cuando deseamos establecer que S |= s , encontrar todos los modelos de S y comprobar que también lo son de s. En su lugar utilizaremos reglas para derivar consecuencias lógicas (conclusiones) de un conjunto de fórmulas (premisas). Estas reglas son conocidas como Reglas de Inferencia.

1.5 <u>REGLAS DE INFERENCIA</u>

Un hecho fundamental en la lógica es la existencia de reglas de inferencia, las cuales son la herramienta para derivar consecuencias lógicas de otras fórmulas. La Inferencia Lógica es el proceso de derivar una fórmula s de un conjunto S de fórmulas aplicando reglas de inferencia, generalmente con el propósito de mostrar que S |= s (s es consecuencia lógica de S).

Podemos intentar derivar la fórmula s directamente, o utilizar la técnica de reducción al absurdo suponiendo not(s) y generando una sucesión de consecuencias lógicas cuyo enunciado final sea la fórmula vacía, generando de esta manera una contradicción.

A continuación se expone una regla de inferencia muy importante.

Cuando las fórmulas con que tratamos están en forma de cláusula nos auxiliamos de una regla de inferencia llamada Regla de Resolución para efectuar la prueba de S |= s por reducción al absurdo. Limitaremos nuestra atención a las cláusulas de Horn (las que contienen a lo más una literal no negada), tratando por lo tanto únicamente con reglas, hechos y metas.

Existen dos casos especiales de la regla de resolución. El primero es el que trata con una meta M y una regla R, aplicandose de la forma siguiente:

```
M :   not(a)
R :   a <- b
------------
M':   not(b)
```

A la cláusula M', la cual es el resultado de aplicar esta regla a las cláusulas M y R, se le conoce como el "resolvente" y al proceso de aplicar la regla como "resolver".

El segundo caso especial de dicha regla trata con una meta M y un hecho H :

```
M :   not(a)
H :   a <-
------------
M':   □
```

El resolvente que se obtiene en este caso es la cláusula vacía (denotada por []), la cual será utilizada para denotar una contradicción o inconsistencia.

Dado que estamos limitados a las cláusulas de Horn, el caso más general de la regla de resolución se presenta cuando resolvemos una meta M con varios predicados y una regla R con varios antecedentes :

$$
\begin{array}{ll}
M: & not(a_1,..,a_n) \\
R: & a_k \leftarrow b_1,..,b_m \qquad (\text{ donde } 1<k<n \text{ }) \\
\hline
M': & not(a_1,..,a_{k-1},b_1,..,b_m,a_{k+1},..,a_n)
\end{array}
$$

El predicado a_k en la meta M coincide con el consecuente de la regla R. El proceso de resolver estas cláusulas sustituye al predicado a_k de la meta, por el antecedente de la regla, derivando de esta forma el resolvente M'.

Notemos que en cada uno de los casos el proceso de resolver consiste en cancelar una literal negada a_i en la meta y reemplazarla por las literales negadas de la cláusula que tiene como consecuente a a_i, generando de esta manera una nueva meta.

Ejemplo:

```
M : not(abuelo(luis,ana))
R : abuelo(luis,ana) <- padre(luis,jose) , padre(jose,ana)
-----------------------------------------------------------
M': padre(luis,jose) , padre(jose,ana)
```

La prueba de S |= s por reducción al absurdo consiste en la aplicación sucesiva de la regla de resolución hasta generar la cláusula vacia, este proceso es conocido como una Refutación.

Dado que las literales que intervienen en las cláusulas son predicados que relacionan a varios términos nos encontramos con el problema de aplicar la regla de resolución a cláusulas de la siguiente forma :

```
M" : not(p(X))          X = variable
H" : p(c)               c = constante
```

En este caso nos auxiliamos de una regla de inferencia llamada Regla de Sustitución la cual establece que :

$\forall X$ p(X) implica p(t) , t es un término

donde **$\forall X$** significa "Para todos los individuos del dominio de interpretación de la cláusula", y en la que **p(t)** representa una cláusula similar a **p(X)** sólo que se han reemplazado las ocurrencias de X por el término t. De esta forma podemos resolver cláusulas como las antes citadas (M" y H") "sustituyendo" la ocurrencia de la variable X por el término t a cada una de ellas, generando así cláusulas idénticas, las cuales derivan la cláusula vacía.

Definimos una Sustitución $\theta = \{V_1/t_1,..,V_n/t_n\}$ como el conjunto de asignaciones de términos t_i a variables V_i, en la que a ninguna variable le es asignado mas de un término. Ejemplo :

```
M : not( d(s(z),X,s(s(z))) )
R : d(s(X'),Y',s(Z')) <- d(X',Y',Z')
--------------------------------------
M': not(d(z,X,s(z)))      θ = { X'/z , Y'/X , Z'/s(z) }
```

En éste caso al aplicar la sustitución θ a las cláusulas M y a R generamos las cláusulas θM y θR,

```
θM :  not( d(s(z),X,s(s(z))) )
θR :  d(s(z),X,s(s(z))) <- d(z,X,s(z))
```

las cuales al ser resueltas generan la nueva meta,

M': not(d(z,X,s(z)))

Al resultado de aplicar una sustitución θ a una cláusula F se le llama una instancia de F y lo denotamos como θF.

Si aplicamos θ a dos cláusulas F1 y F2 y llegamos a la misma instancia (llamada "instancia común"), a θ le llamamos un unificador de F1 y F2, y decimos que las cláusulas son unificables.

En ocasiones, cuando tratamos de unificar un conjunto de cláusulas es posible encontrar más de un unificador para ellas. Ejemplo: Sea

S = { p(f(X),Z) , p(Y,a) }

un conjunto de cláusulas, y θ_1, θ_2 las sustituciones

θ_1 = { Y/f(a) , X/a , Z/a }

θ_2 = { Y/f(X) , Z/a }

Encontramos que θ_1 es un unificador para S pues al aplicarlo obtenemos la instancia común

θ_1(p(f(X),Z)) = p(f(a),a)

θ_1(p(Y,a)) = p(f(a),a)

y lo mismo sucede con θ_2

θ_2(p(f(X),Z)) = p(f(X),a)

θ_2(p(Y,a)) = p(f(X),a)

Observando el número de asignaciones de términos a variables que cada uno de los unificadores realiza, notamos que θ_2 es el que necesita menos asignaciones para unificar al conjunto de cláusulas. En este caso decimos que θ_2 es un unificador mas general que θ_1.

Cuando tratemos de unificar un conjunto S de cláusulas nos auxiliaremos del algoritmo de unificación el cual, si el conjunto es unificable, encuentra la sustitución con el número mínimo de asignaciones, a la que llamaremos el Unificador lo Mas General de S. Daremos como un hecho la existencia de este algoritmo y se recomienda consultar una descripción detallada en el libro "Foundations of Logic Programming" de J.W. Lloyd[1].

Hemos dado una descripción de la forma de aplicar la regla de resolución a un par de cláusulas de Horn con el propósito de generar una nueva cláusula, la cual es consecuencia lógica de estas. Con este propósito nos apoyamos en la regla de sustitución para poder asignar términos a las variables de las cláusulas, y del algoritmo de unificación para poder encontrar el unificador lo más general que nos permita, al aplicarlo a las fórmulas, resolverlas e inferir la nueva cláusula.

Podemos ver ahora al conjunto de cláusulas de Horn que tienen a lo más una literal no negada (cláusulas de programa), como un "lenguaje de programacion", y a un programa lógico (un conjunto finito de cláusulas de programa) como un "programa".

En esta interpretación se utiliza como única regla de inferencia a la regla de resolución.

1.6 INTERPRETACION PROCEDURAL

Podemos ver a las cláusulas de Horn como un lenguaje de programación si interpretamos cada cláusula como la definición de un procedimiento. El consecuente o cabeza de la cláusula será el nombre del procedimiento y el antecedente representará al cuerpo del procedimiento.

Una cláusula meta

$$\leftarrow P_1, \ldots, P_n$$

es vista como un programa principal o procedimiento inicial que no tiene nombre, es decir, un conjunto de condiciones por satisfacer que no tienen conclusiones. Trataremos de satisfacer las condiciones invocando como procedimientos uno a uno los predicados (submetas) que conforman el cuerpo de la meta.

Los procedimientos son invocados por unificación, es decir, dada una llamada a un procedimiento buscamos una cláusula en el programa cuya cabeza sea unificable con el procedimiento al que se llama. Aplicamos el unificador al procedimiento que llama y al procedimiento seleccionado, generando una instancia común de estos (haciéndolos sintácticamente idénticos), y reemplazamos la llamada original por el cuerpo del procedimiento seleccionado.

La ejecución termina cuando todos los procedimientos de la cláusula meta inicial han sido satisfechos. Mostraremos a continuación un ejemplo :

Sea S el conjunto de cláusulas siguientes

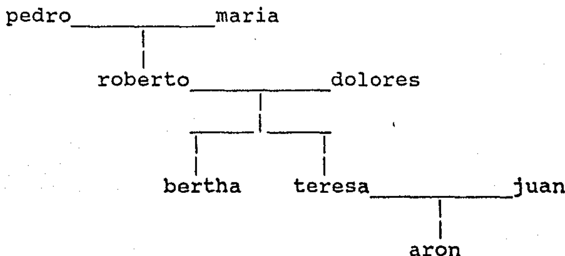
```
    abuelo(X,Y) <- es_un_padre(X,Z) , es_un_padre(Z,Y)

    es_un_padre(X,Y) <- padre(X,Y)
es_un_padre(X,Y) <- madre(X,Y)

    padre(pedro,roberto) <-
    padre(roberto,bertha) <-
    padre(roberto,teresa) <-
    padre(juan,aron) <-

    madre(maria,roberto) <-
    madre(dolores,bertha) <-
    madre(dolores,teresa) <-
    madre(teresa,aron) <-
```

las cuales representan el árbol familiar siguiente :



Supongamos que deseamos saber si aron tiene un abuelo. La cláusula meta o programa principal sería

```
<- abuelo(G,aron)
```

Buscamos todas las cláusulas que tienen al predicado <u>abuelo</u> como consecuente (hay sólo una):

```
abuelo(X,Y) <- es_un_padre(X,Z) , es_un_padre(Z,Y)
```

Hacemos la sustitución { X/G , Y/aron } y reemplazamos la llamada por el cuerpo del procedimiento <u>abuelo</u>, obteniendo la nueva meta

```
<- es_un_padre(G,Z) , es_un_padre(Z,aron)
```

Tenemos ahora dos cláusulas es_un_padre. Podemos escoger trabajar con cualquiera de ellas pero escogeremos la segunda (frecuentemente es más eficiente seleccionar una llamada en la cual hay más argumentos con valores asignados). Necesitamos ahora una cláusula que defina al predicado es_un_padre. Hay dos:

```
es_un_padre(X,Y) <- madre(X,Y)
es_un_padre(X,Y) <- padre(X,Y)
```

Escogamos la primera y hagamos la sustitución { X/Z , Y/aron }, reemplazando la llamada y obteniendo la nueva meta

```
<- es_un_padre(G,Z) , madre(Z,aron)
```

Seleccionemos la segunda llamada y tratemos de encontrar una definición de madre que pueda ser unificada con la llamada seleccionada. Hay solamente una elección posible, la cláusula madre(teresa,aron) <- , y la sustitución que necesitamos es { Z/teresa }. Dado que el cuerpo de la cláusula es vacío (es un hecho), obtenemos la nueva meta

```
<- es_un_padre(G,teresa)
```

Usando la primera definición de es_un_padre y aplicando la sustitución { X/G , Y/teresa} obtenemos la nueva meta

```
<- madre(G,teresa)
```

Finalmente, seleccionamos la cláusula madre(dolores,teresa) y aplicando la sustitución { G/dolores } y resolviendo obtenemos la cláusula vacía.

La respuesta a nuestra meta original la encontramos en las sustituciones hechas a lo largo del proceso (inferencia lógica). en este caso hemos demostrado que dolores, la asignación hecha a G de la llamada original, es un abuelo de aron.

Es importante notar que escogimos libremente el orden para ejecutar las llamadas en una meta (selección de submetas), y que de la misma forma tuvimos libertad de escoger cualquier definición de procedimiento cuyo nombre pueda ser unificado con la submeta seleccionada (selección de cláusulas). En este sentido, un lenguaje de programación lógica basado en cláusulas de Horn es no-determinista.

La meta original y la selección de submetas en cada paso determinan un Espacio de Búsqueda conteniendo varios caminos posibles. Si hubiésemos seleccionado un orden de ejecución diferente pudimos haber demostrado que roberto es también abuelo de aron.

Observemos que la definición de abuelo no implica que siempre debamos dar un nieto y tratar de encontrar un abuelo. Podemos usar la misma definición y preguntar por un nieto de un abuelo dado. Ejemplo :

```
<- abuelo(pedro,X)
```

También podemos dar dos nombres y preguntar si pedro es un abuelo de bertha

```
<- abuelo(pedro,bertha)
```

O inclusive no dar ningún nombre y preguntar cuantos elementos del dominio de interpretación se encuentran relacionados por el predicado abuelo.

```
<- abuelo(X,Y)
```

Hacemos referencia a esta flexibilidad para la forma de ejecutar llamadas a procedimientos como la Invertivilidad de los programas lógicos.

Las características mencionadas anteriormente le dan un gran poder a la programación basada en cláusulas de Horn y por lo mismo mas dificultades para su implementación como un lenguaje de programación. En el siguiente capítulo examinaremos a un lenguaje de programación lógica determinístico llamado PROLOG (por PROgramming in LOGic), el cual existe en varias versiones y para diferentes máquinas.

CAPITULO 2

P R O L O G

Prolog fué diseñado en 1972 en Marsella, Francia, por el grupo de investigación de Alain Colmerauer. El objetivo original de este trabajo fué la integración del principio de resolución como única regla de inferencia en un lenguaje de programación, en lugar de las múltiples reglas propuestas por los lógicos. Esto facilitó el diseño de un lenguaje de programación que permite al programador hacer una simulación del proceso de razonamiento, haciendo deducciones de la información contenida en fórmulas lógicas.

2.1 PROLOG COMO UN LENGUAJE DE PROGRAMACION LOGICA

Prolog es un lenguaje determinístico basado en cláusulas de Horn. La sintaxis de las cláusulas en Prolog es la misma que la de las cláusulas de Horn, excepto que cada cláusula debe ser terminada con un punto y el símbolo :- sustituye al de implicación en una regla.

Siguiendo estas reglas de sintaxis las cláusulas de Horn quedan expresadas en un programa escrito en Prolog de la siguiente forma :

A.	Hechos
A :- $B_1, \ldots, B_n$.	Reglas
:- $B_1, \ldots, B_n$.	Metas

El hecho de que Prolog es un lenguaje determinístico implica que existe un orden de ejecución impuesto al lenguaje. Este orden de ejecución es conocido como "Estrategia Estandard" y consiste en una regla para la selección de llamadas a procedimientos, llamada Regla de Computación, y de una regla para la selección de procedimientos o Regla de búsqueda.

REGLA DE COMPUTACION . Dada una meta, selecciona siempre la primera llamada a procedimiento (submeta). Ejemplo:

Sea :- $B_1,B_2,...,B_n$ una meta. Siguiendo la regla de computación, B_1 será la llamada a procedimiento que será seleccionada.

REGLA DE BUSQUEDA . Dada una meta, selecciona siempre los procedimientos (cláusulas) en el orden de aparición en el programa. Ejemplo:

Sea el siguiente conjunto de cláusulas un programa, y B_1 la submeta seleccionada por la regla de computación.

(1) B_2 :- M_1,M_2,M_3

(2) B_1 :- M_3

(3) B_1 :- M_5,M_6

Siguiendo la regla de búsqueda, la cláusula (2) será la seleccionada para únificarse con la submeta B_1.

ESTRATEGIA ESTANDARD . Es la combinación de la regla de computación y la regla de búsqueda.

Armado con la estrategia estandar y la regla de resolución, el intérprete de Prolog trata de encontrar todas las soluciones a una meta planteada, inspeccionando el espacio de búsqueda sistemáticamente.

Durante la ejecución de un programa lógico, cuando es generada la cláusula vacía (derivación exitosa, denotada □), o cuando se ha llegado al punto en que ningún procedimiento responde a la llamada seleccionada (derivación fallida, denotada por ■), el intérprete se auxilia de un principio llamado "Backtrack" o también "Retroceso", el cual le permitirá recorrer el resto del espacio de búsqueda tratando de encontrar otras soluciones (en el caso de una derivación exitosa), o una solución (en el caso de una derivación fallida).

El principio del backtrack consiste en regresar a la última llamada activada y buscar otros procedimientos que puedan ser unificados con esta, recorriendo así, los distintos caminos que puedan existir en el espacio de búsqueda. Revisemos este principio.

Supongamos que tenemos una meta inicial como la siguiente:

$:- M_1, \ldots, M_i, \ldots, M_n$

y también que la estrategia estandard nos dice que la llamada activa es M_i (lo que implica que $M_1, \ldots, M_{i-1}$ han sido resueltas exitosamente, aunque no necesariamente se han recorrido todas las ramas del espacio de búsqueda).

En general pueden existir varios procedimientos unificables con M_i (llamados los candidatos de M_i), y también es posible que algunos de ellos ya hayan sido seleccionados con el propósito de resolverlos. Sean P,P',P",etc los restantes en orden de aparición en el programa.

Por la estrategia estandard seleccionamos P

$P :- Q_1, \ldots, Q_m$

Lo cual inicia un intento de resolver la llamada, hecho que puede terminar exitosamente o fallar.

Si la ejecución resuelve todas las llamadas en P, el intérprete hace una salida exitosa de P y transfiere el control a M_{i+1} la cual será la siguiente llamada activada.

Por otro lado, si es imposible resolver alguna de las llamadas en P, el intérprete hace una salida fallida de P y transfiere el control al siguiente candidato de M_i, en este caso P'. Este comportamiento es llamado Backtrack después de falla.

Si después de esta operación de backtrack ninguno de los candidatos restantes de M_i puede ser resuelto, será necesaria una nueva operación de backtrack con el propósito de encontrar nuevas opciones para resolver la llamada M_{i-1} o sus antecesores.

Si la ejecución resuelve todas las llamadas de la meta original, entonces el intérprete reporta la solución encontrada y transfiere el control a la llamada más reciente, para la cual, algunos candidatos permanecen sin activar. Este procedimiento es conocido como Backtrack después de éxito.

Daremos a continuación, un ejemplo del uso de la estrategia estandard y del principio del backtrack, tomando como programa el conjunto de cláusulas siguiente, las cuales identificaremos por el número de cláusula que le asociamos. Para hacer más claro el ejemplo en todos los casos se subraya la submeta seleccionada.

La submeta c es seleccionada y como no existe alguna cláusula que pueda ser unificada con ella el intérprete hace una salida fallida de c. Realiza una operación de backtrack, regresando el control a la submeta f seleccionada anteriormente y busca otra cláusula con cual unificarla. Al no encontrarla, realiza otra operación de bactrack regresando el control a la submeta b y busca unificarla con otra cláusula. Como no existe otra cláusulas con la cual unificarla, se realiza una nueva operación de backtrack, regresando el control a la submeta a seleccionada inicialmente y busca otra cláusula con cual unificarla siendo esta la número 2.

:- c , d , e	Falla, Backtrack a f
:- f , c , d , e	Falla, Backtrack a b
:- b , c , d , e	Falla, Backtrack a a
:- a , e	

Meta originada	Procedimiento seleccionado
:- e , f , e.	2

Al seleccionar la submeta e la unifica con la cláusula 4 y sustituyendo su cuerpo por e genera la nueva meta.

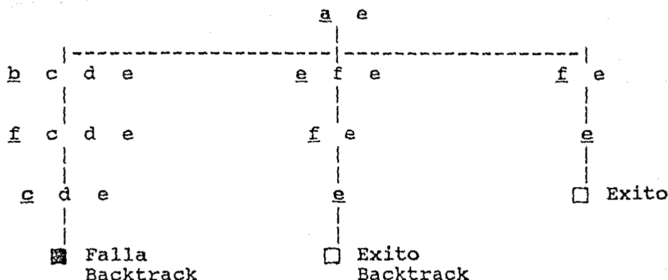
Meta originada	Procedimiento seleccionado
:- f , e.	4

Seleccionando f y unificando con la cláusula 5 que tiene cuerpo vacío genera una nueva meta.

Meta originada	Procedimiento seleccionado
:- e.	5

Seleccionando la submeta e y unificando con la cláusula 4, la cual tiene cuerpo vacío, se genera la cláusula vacía y puesto que no existen mas submetas la meta es resuelta exitosamente.

Meta originada	Procedimiento seleccionado
:- □	4 (Exito)



Esta representación en forma de árbol del espacio de búsqueda nos muestra de una manera más clara el proceso de resolución de la meta original.

Hemos examinado el proceso de ejecución de un programa lógico por un intérprete de Prolog, ahora examinaremos las estructuras de datos que el lenguaje soporta, las cuales son bastante diferentes a las de otros lenguajes de programación.

2.2 ESTRUCTURAS DE DATOS.

Como se dijo antes, un programa escrito en Prolog (un Programa Lógico) es un conjunto finito de Cláusulas (cláusulas de programa) de la forma P_0 :- $P_1,\ldots,P_n$, donde cada P_i es llamado un Procedimiento o Predicado y tiene la forma $P(t_1,\ldots,t_m)$. P es llamado un Símbolo predicado y $t_1,\ldots,t_m$ son Términos. Los términos pueden ser Constantes, Objetos compuestos o Variables, los cuales se describen a continuación.

CONSTANTES.

Las constantes son cadenas de caracteres que representan un objeto único (se representan a ellas mismas) el cual no tiene estructura, por lo que son llamados "átomos".

Los átomos pueden ser usados para representar distintas clases de información, tales como nombres de ciudades, números telefónicos, edades, precios, colores, etc.

La interpretación de los átomos es labor del programador, pues como un ejemplo, el átomo 1990 puede significar el precio de un artículo, el año actual, o la distancia entre dos ciudades.

OBJETOS COMPUESTOS O ESTRUCTURAS.

Son agrupaciones complejas de objetos que pueden ser manejados en una estructura (como un solo objeto), en lugar de hacerlo separadamente.

Los elementos de la estructura pueden ser cualquier tipo de dato, por lo que pueden representar objetos muy complejos. Las estructuras tienen una notación muy similar a la usada para describir funciones en matemáticas y consiste de un nombre de estructura seguido de una descripción de sus componentes separados por comas y delimitados por paréntesis. Al nombre de la estructura se le conoce como Functor y a sus componentes como Argumentos.

Por ejemplo, una descripción de los elementos del agua puede ser puesta en la estructura:

```
agua(hidrogeno,oxigeno)
```

La descripción de un registro de alumno :

```
alumno( nombre(oscar,valencia),
        direccion(calle(tlalpan,4567),col(portales)),
        telefono(5509300),carrera(matematicas) )
```

En los ejemplos anteriores agua y alumno son functores, llamados también functores principales y definen la estructura general del objeto compuesto, nombre y carrera son argumentos de alumno que también son functores, hidrogeno y oxigeno son constantes.

Cuando deseamos escribir una expresión aritmética, Prolog nos facilita una serie de functores predefinidos los cuales son :

+ , - , * , / , mod

con los cuales podemos escribir estructuras de una forma más compacta, de acuerdo con su significado usual. Ejemplo :

```
suma(2,3)                        2+3

multiplica(2,suma(5,1))          2*5+1
```

Observemos que la sintaxis que utilizamos para las estructuras y predicados es similar, sólo que las estructuras son objetos (nos ofrecen una forma de construcción de nombres más elaborada), mientras que los predicados tienen una interpretación procedural (son la llamada o la cabeza de un procedimiento).

[H|T]	-Todas las listas no vacias.
[Var1,Var2,Var3]	-Todas las listas de tres elementos.
[V1,V2,xxx]	-Las listas de tres elementos tal que el tercero es igual a la constante xxx.
alumno(X,fisica)	-Todos los alumnos de física.
Variablex	-El conjunto de todos los objetos, puesto que una variable puede ser instanciada con cualquier objeto.

2.3 ESTRUCTURAS DECLARATIVAS.

Existen únicamente cinco conceptos sintácticos en la notación de Prolog: tres conceptos, que como ya mencionamos anteriormente comprenden a los Términos (constantes, variables y funciones), y otros dos conceptos que están formados por agrupaciones de mayor nivel (Cláusulas y Procedimientos) los cuales se describen a continuación.

CLAUSULAS.

Son cláusulas de Horn (hechos, reglas y metas) y tienen la forma P_0 :- $P_1,...,P_n$, donde cada P_i es llamado un procedimiento o predicado. Cada P_i tiene la forma $P_i(X_1,...,X_m)$,donde X_i es un término.

Desde un punto de vista procedural las cláusulas pueden ser interpretadas como la definición de un procedimiento en la que el predicado P_0 puede ser visto como el encabezado del procedimiento y la lista de predicados $P_1,..,P_n$ como el cuerpo del procedimiento. Los predicados que conforman el cuerpo de P_0 pueden ser vistos como llamadas a otros procedimientos donde los términos $X_1,..,X_m$ toman el papel de los parámetros. Ejemplo :

```
abuelo(X,Y):-padre(X,Z),padre(Z,Y).

es_un(Animal,tigre):-es_un(Animal,mamifero),
                     es_un(Animal,carnivoro),
                     es_un(Animal,"con rayas").

tiene_vida(Planeta):-distancia_al_sol(Planeta,93000000),
                     atmosfera(Planeta),
                     satelites(Planeta,1).
```

2.4 COMPORTAMIENTO COMPUTACIONAL.

En el capítulo I vimos como es posible dar una interpretación procedural a los programas lógicos, ahora describiremos la forma de simular en Prolog las clases más importantes de estructuras computacionales típicas y analizaremos la forma en que un programa lógico se comporta en estos casos.

PROCESAMIENTO DE DATOS.

El procesamiento de datos en general consiste en suministrar datos (entrada) a un proceso , el cual nos regresa en respuesta otros datos (salida), los cuales han sido generados en función de los datos de entrada. En Prolog este proceso se ve reflejado al momento que se da la llamada a un procedimiento.

Sea $p(t_1,..,t_k)$ la llamada a un procedimiento. Como las t_i son términos, es posible que algunos de ellos sean variables. Sean $X_1,..,X_m$ esas variables. De la misma forma, la cabeza del procedimiento seleccionado puede contener variables, sean estas $Y_1,..,Y_n$.

Cuando el procedimiento es invocado, el unificador generado asigna términos a las variables X_i y Y_i. Los términos asignados a las variables Y_i son el equivalente a los datos de entrada, mientras que los asignados a las variables X_i equivalen a los datos de salida. Ejemplo, sean:

```
:-sucesor(3,N),predecesor(N,M).        : Meta

  sucesor(X,Y) :- Y is X+1.            : Procedimiento
```

y sea sucesor la cláusula que se unifica con la primera llamada a procedimiento de la meta.

El unificador $\theta = \{ X/3 , N/Y \}$ al ser aplicado asigna términos a la cabeza del procedimiento invocado (entrada 3) y a la llamada del procedimiento en la meta (salida Y), sustituyendo también las ocurrencias de las variables asignadas que aparecen en el resto de la meta así como en el cuerpo del procedimiento.

```
:-sucesor(3,N),predecesor(N,M).

  sucesor(X,Y) :- Y is X+1
```

Para añadir dos listas utilizamos el procedimiento append.

```
append([],Lista,Lista).
longitud([H|T],L,[H|T1]) :- append(T,L,T1).
```

El cual establece en su primera cláusula que: "Añadir la lista vacía con cualquier lista da como resultado la misma lista",y en la segunda que "Dadas las listas [H|T] y L el resultado de añadirlas será la lista [H|T1] si al añadir T y L se obtiene T1".

Para determinar recursivamente, si un valor está contenido en una lista, el predicado miembro compara el valor con la cabeza de la lista,

```
miembro(X,[X|_]).
miembro(X,[_|Y]) :- miembro(X,Y).
```

La primera cláusula establece que "X es miembro de una lista si es igual a la cabeza de la lista", mientras que la segunda establece que "X es miembro de una lista si es miembro del resto de la lista".

ITERACION.

Existe un caso especial de los procedimientos recursivos llamados procedimientos iterativos, los cuales ejecutan repetidamente un segmento de programa; cada repetición o paso iterativo ejecuta el segmento de programa completamente antes de que el siguiente paso se inicie. Podemos ver un procedimiento recursivo como uno iterativo cuando la llamada recursiva es la ultima en el cuerpo del procedimiento.

Hasta este momento hemos revisado las características que Prolog hereda de la programación lógica, las cuales son llamadas características "puras". A continuación veremos algunas características "impuras" que han sido añadidas al lenguaje para hacerlo mas práctico.

2.5 CARACTERISTICAS EXTRALOGICAS.

Prolog tiene varias propiedades extras a las de la programación lógica, las cuales facilitan la escritura de programas eficientes.

El corte en este conjunto de cláusulas asegura que sólo una tarea será realizada en función del término asignado a la variable Temp.

2) Cuando se desea limitar el espacio de búsqueda, es decir, sólo queremos una respuesta.

```
intersección(L1,L2) :- miembro(E,L1),
                       miembro(E,L2),!.
```

El propósito del procedimento intersección es encontrar un elemento común en las listas L1 y L2. La inclusión del corte en este caso, evita la generación de elementos de L1 una vez que ha sido encontrado el primer elemento común.

3) En combinación con el predicado fail (el cual siempre falla) para implementar la negación por falla, que se ve a continuación.

NEGACION POR FALLA.

Prolog responde "no" a una meta cuando falla al tratar de resolverla. La negación por falla puede entonces ser interpretada como "Si no puedo probarlo, debe ser falso".

El operador not que representa la negación por falla (en lugar de la negación lógica) puede ser implementado con las cláusulas,

```
not(X) :- X , ! , fail.
not(_).
```

La primera cláusula intenta satisfacer el argumento X de not. Si la meta X sucede, entonces el corte y fail son alcanzados. El predicado fail forza la falla, y el corte evita la consideración de la segunda cláusula. Por otro lado, si la meta X falla, entonces la segunda regla sucede pues la variable anónima _ se unifica con cualquier término.

Esta forma de interpretar la negación es aceptable en aplicaciones como las bases de datos, donde podemos afirmar que nuestro conocimiento acerca de los datos es completo; es decir, Conocemos que información está en la base de datos, pero no nos interesa enumerar la infinidad de información que no está almacenada en ella (lo cual sería equivalente a almacenar información "negativa").

Hemos dado una descripción del lenguaje de programación Prolog, mostrando las características que lo clasifican como un lenguaje lógico. Examinamos también las características extralógicas que ofrece, con el propósito de mejorar la eficiencia al momento de ejecución. Estamos ahora en posición de estudiar algunas de las áreas en las que Prolog tiene una aplicación directa, como : Desarrollo de Sistemas expertos, Representación del Conocimiento , procesadores de lenguaje natural , sistemas manejadores de bases de datos, etc...

Nuestro interés en este momento se encuentra enfocado hacia el área de Manejadores de Bases de Datos, por lo que en los siguientes capítulos examinaremos las propiedades que debe cumplir una aplicación para ser considerada un manejador de bases de datos relacional, y describiremos la forma de implementar estas propiedades haciendo uso del lenguaje Prolog.

3) Un Lenguaje de manipulación de datos tan poderoso, al menos, como el Algebra Relacional.

A continuación describiremos al modelo de datos relacional y sus propiedades.

3.2 CONCEPTOS BASICOS DEL MODELO DE DATOS RELACIONAL.

Esencialmente un modelo de datos provee de mecanismos de estructuración y operacionales, para soportar el manejo de datos. En esta sección describiremos estos mecanismos haciendo énfasis en el modelo relacional de datos.

ESTRUCTURAS.

Los mecanismos de estructuración que ofrece el modelo de datos relacional, nos permiten agrupar datos elementales (Atomos) en datos compuestos (Estructuras). Un número fijo de átomos puede ser agrupado para formar objetos estructurados (Eneadas), y estos a la vez, se agrupan formando Relaciones. Definamos entonces una relación.

Un Dominio es un conjunto de valores de tipo similar, por ejemplo, los siguientes conjuntos son dominios:

- El conjunto de números de parte de un inventario.
- El conjunto de materias que forman un plan de estudios.
- El conjunto de nombres de profesores de una escuela.

Sean D1,...,Dn (n>0) Dominios. Se Define el Producto Cartesiano D1 X ... X Dn como el conjunto de todas las eneadas (t1,...,tn) tales que ti es elemento de Di.

Una Relación en estos n dominios es definida como un subconjunto del producto cartesiano D1 X ... X Dn y se dice que tiene una aridad n.

Notemos que dado que una relación es un conjunto, no existe un orden entre las eneadas que la forman y que, contrariamente, los dominios de donde toman sus valores las eneadas de una relación tienen un orden definido entre ellos (el i-ésimo elemento en cada eneada pertenece al i-ésimo dominio).

Las relaciones resultantes dependen de la operación de derivación en dos formas:

- Sus valores cumplen con un criterio de selección que es determinado por la operación de derivación.

- Su estructura es derivada explícita o implícitamente por la operación de derivación.

Las operaciones efectuadas en una base de datos relacional que generan como resultado relaciones son llamadas Consultas (Queries).

Cuando en una relación existe un atributo que tiene un valor distinto para cada una de sus eneadas, el cual puede ser usado para identificar a las eneadas de la relación, a ese atributo especial se le conoce como llave-primaria.

Aunque no todas las relaciones tienen una llave formada por un único atributo, es posible construir una llave compuesta concatenando atributos, donde el caso extremo sería una llave formada por todos los atributos de la relación. Lo anterior es garantizado por el hecho de que las relaciones son conjuntos y por lo tanto no se permite la repetición de elementos.

Es importante notar que la llave primaria es no-redundante en el sentido de que esta formada por el mínimo de atributos que garantizan la propiedad de identificación única, es decir, no contiene atributos superfluos.

En ocaciones en una relación existe más de una llave, a las cuales se les llama llaves-candidato . Cuando una llave-candidato no es la llave-primaria, se le conoce como llave-alterna.

Cuando un atributo de una relación hace referencia a la llave primaria de otra relación, se dice que es una llave-foránea; Entonces, las llaves primarias y foráneas deben tener el mismo nombre (cuando es posible) y ser del mismo tipo de dato.

Todas las inserciones, actualizaciones y bajas de las relaciones base están restringidas por las siguientes dos reglas:

Integridad de entidades.- Ninguna llave primaria puede ser nula o tener un componente nulo.

Integridad Referencial.- Una llave foránea debe ser nula o ser igual a un valor de la llave primaria correspondiente.

Como un ejemplo de los conceptos citados anteriormente, pensemos en una base de datos que agrupa información de una escuela. Haciendo uso de un supuesto mecanismo para definir estructuras creamos la relación "alumno" y definimos los dominios de sus atributos.

```
Nombre de la
relación                  Atributos   Dominios
   |                          |           |
   |                          |           |
alumno  =  relacion  cvealu   :   clave ;
                     nomalu   :   nombre ;
                     caralu   :   carrera ;
                     semalu   :   semestre
           fin ;

clave    =  string (5) ;
nombre   =  string (35) ;
carrera  =  ("BIOLOGIA","MATEMATICAS","FISICA",
             "ACTUARIA") ;
semestre =  1..10
```

La relación alumno representada tabularmente tiene la forma siguiente:

cvealu	nomalu	caralu	semalu
23453	MARIA ELENA TREJO	MATEMATICAS	3
24532	LAURA RODRIGUEZ	BIOLOGIA	7
74357	MONICA SUAREZ	BIOLOGIA	4
97263	OSCAR MOLINA	FISICA	6
96634	JULIO CORONA	MATEMATICAS	5
.	.	.	.
.	.	.	.

Cuando tratamos de aplicar un mecanismo de actualización a esta relación, las restricciones impuestas a las relaciones al momento de su definición, cuidarán la integridad de la información. Por ejemplo: No será posible insertar una eneada que en el atributo carrera tenga un valor distinto a los especificados en el dominio del atributo, como tampoco sería posible insertar una eneada que ya existe en la relación duplicándola.

Cabe aclarar que una base de datos requiere que el usuario conozca qué atributos agrupar para formar relaciones, las cuales serán operadas haciendo uso de los mecanismos de estructuración y operación.

3.3 ALGEBRA RELACIONAL.

El Algebra Relacional es definida como una colección de operaciones sobre relaciones que tiene como operandos a una o más relaciones y produce como resultado otra relación. A este hecho se le conoce como la cerradura de las relaciones con respecto al álgebra relacional.

Los operandos de cualquier operación del álgebra relacional pueden ser relaciones o expresiones que al ser evaluadas generan relaciones, es decir, las expresiones pueden ser anidadas.

En lo que sigue R,S denotan relaciones; A, B1, B2, C denotan colecciones de atributos ; c es una eneada de grado (número de columnas) y dominios apropiados al ejemplo.

Las operaciones básicas del álgebra relacional son las siguientes :

UNION.

La Unión de relaciones R y S, denotada R U S, es el conjunto de eneadas en R, en S o en las dos. Es importante aplicar el operador unión a relaciones de la misma aridad, también llamadas Unión-Compatibles.

INTERSECCION.

La Intersección de relaciones R y S, denotada R ÿ S, es el conjunto de eneadas en R que también lo son de S.

DIFERENCIA.

La Diferencia de las relaciones R y S, denotada R - S, es el conjunto de eneadas en R pero no en S.

PRODUCTO CARTESIANO.

El Producto Cartesiano de las relaciones R y S, denotado R X S , es el conjunto de eneadas resultantes de concatenar una eneada r de R con una eneada s de S. Por lo anterior, si la aridad de R es m y la de S es n, el producto cartesiano tiene aridad m+n, ya que los primeros m componentes de sus eneadas forman una eneada de R y sus n componentes restantes forman una eneada de S.

SELECCION-THETA.

Sea θ una de las relaciones binarias <, <=, =, >=, >, $\neq$ que es aplicable al atributo A y la eneada c. Entonces R [A θ c] es el conjunto de eneadas de R, cuyos A-componentes están en la relación θ con la eneada c. En lugar de la eneada c, otro atributo B puede ser citado, siempre que A y B estén definidos en dominios comunes. Por lo que R [A θ B] es el conjunto de eneadas en R que satisfacen la condición de que su A-componente está en la relación θ con su B-componente. Cuando θ es la igualdad, el operador Selección-Theta es llamado simplemente SELECCION. Ejemplos:

R

A	B	C
p	1	2
p	2	1
q	1	2
r	2	5
r	2	3

R [B > C]

A	B	C
p	2	1

R [A $\neq$ r]

A	B	C
p	1	2
p	2	1
q	1	2

R [A = r]

A	B	C
r	2	5
r	2	3

PROYECCION.

Es la relación R [A1,..,An], que se obtiene al borrar todas las columnas de R, excepto aquellas especificadas por A1,..,An, y toma un subconjunto de las eneadas resultantes, eliminando las eneadas repetidas. La Proyección también nos permite reordenar las columnas resultantes de la operación. Ejemplos:

R

A	B	C
p	1	2
p	2	1
q	1	2
r	2	5
r	2	3

R [A,B]

A	B
p	1
p	2
q	1
r	2

R [B,C]

B	C
1	2
2	1
2	5
2	3

R [B]

B
1
2

UNION-THETA.

Dadas las relaciones R(A,B1) y S(B2,C), con B1 y B2 definidos en un dominio común, sea θ una de las relaciones binarias =, <, <=, >=, >, ≠ la cual es aplicable al dominio de los atributos B1, B2. La Unión-Theta de R en B1 con S en B2, denotado por R [B1 θ B2] S, se obtiene concatenando los renglones de R con los renglones de S siempre que el B1-componente del renglón de R esté en la relación θ con el B2-componente del renglón de S. Cuando θ es la igualdad, el operador es llamado UNION-EQUIVALENCIA. Notemos que la relación unión-equivalencia tendrá dos columnas idénticas (una derivada de B1 y la otra de B2). Ejemplos:

R (A B C)

A	B	C
p	1	2
p	2	1
q	1	2
r	2	5
r	3	3

S (D E)

D	E
2	u
3	v
4	u

R [C = D] S (A B C D E)

A	B	C	D	E
p	1	2	2	u
q	1	2	2	u
r	3	3	3	v

R [C > D] S (A B C D E)

A	B	C	D	E
r	3	3	2	u
r	2	5	2	u
r	2	5	3	v
r	2	5	4	u

Si las relaciones a las que se les aplica el theta-unión tienen nombres de atributos en común, los nombres de atributos de la relación resultante deben ser especificados, así como los atributos que están en la relación binaria deberán ser etiquetados con el nombre de la relación. Ejemplo:

T (D,E,F,G) = R(A,B) [R.B > S.B] S(A,B)

UNION NATURAL.

Es igual al unión-equivalencia, sólo que las columnas redundantes generadas son removidas. Sean R y S las relaciones descritas anteriormente, la Unión Natural, denotado como R [C * D] S, es:

R [C * D] S (A B C E)

A	B	C	E
p	1	2	u
q	1	2	u
r	3	3	v

DIVISION.

Dadas las relaciones R(A,B1) y S(B2) con B1 y B2 definidos en el mismo dominio, entonces R [B1 - B2] S es el máximo subconjunto de R [A] tal que su producto cartesiano con S(B2) está contenido en R. Ejemplo:

R (A B)

A	B
p	1
p	2
p	3
q	1
r	1
r	3

S (C)

C
1
3

R [B - C] S (A)

A
p
r

3.4 <u>CALCULO RELACIONAL</u>.

Un lenguaje de manipulación de datos, tan poderoso como el Algebra Relacional, puede ser obtenido al tomar un subconjunto del cálculo de Predicados llamado Cálculo Relacional, pensando exclusivamente en bases de datos relacionales..

Cuando escribimos una expresión en álgebra relacional, se proporciona una secuencia de operaciones (en el caso de expresiones anidadas) que generan la relación que responde a la consulta. En el caso del cálculo relacional se da una descripción formal de la información deseada sin especificar cómo obtenerla.

Existen dos formas de cálculo relacional: una en la que las variables representan eneadas, y otra en la que las variables representan valores de dominios. La primera se conoce como Cálculo Relacional de Eneadas y la otra como Cálculo Relacional de Dominios. Los dos tipos son muy similares, por esta razón sólo trataremos el cálculo relacional de eneadas.

CALCULO RELACIONAL DE ENEADAS.

Uno de los aspectos importantes en el cálculo relacional de eneadas es la noción de Variable-eneada, esto es, una variable cuyos valores permitidos son eneadas de una relación.

La base para la construcción de consultas en este tipo de cálculo relacional es la Expresión. Las expresiones en el cálculo relacional de eneadas están formadas por los siguientes tres elementos:

1) Variables-eneada.

Sean T, U, V,... variables eneada , donde cada variable toma sus valores de las eneadas de una relación. Si A es un atributo o conjunto de atributos, T.A representa el A-componente de T.

2) Condiciones.

Tienen la forma x * y , donde * es uno de los operadores =, ≠, <, >, <=, >=, al menos uno de los operandos x o y es una expresión de la forma T.A y el otro puede ser una expresión similar o una constante.

3) Fórmulas bien formadas.

Son construidas con condiciones, operadores booleanos (AND, OR, NOT), y cuantificadores (∃,∀) de acuerdo con las reglas:

- Toda condición es una fórmula
- Si f y g son fórmulas, también la son:
  (f), NOT(f), (f AND g), (f OR g)
- Si f es una fórmula en la cual t ocurre libre (fuera del alcance de un cuantificador), entonces, ∃t(f) y ∀t(f) son fórmulas.

Podemos ahora presentar la forma general que tiene una expresión en al cálculo relacional de eneadas.

T.A,U.B,...,V.C [TAL_QUE F]

donde T, U,...,V son variables-eneada, A, B,...,C son atributos de sus relaciones asociadas y F es una fórmula que contiene a T, U,...,V como variables libres. El resultado de evaluar esta expresión es un subconjunto del producto cartesiano T X U X...X V para cuyas eneadas F es verdadera. Ejemplo:

Sean LECTOR y LIBRO relaciones con los atributos NOMBRE, CVE_TEXTO, EDAD, NIVEL_INGRESO, PRECIO, AREA; y las variables-eneada correspondientes V_LEC y V_LIB.

L E C T O R:

NOMBRE	CVE_TEXTO	EDAD	NIVEL_INGRESO
MARIA ELENA TREJO	MAT123	19	9
LAURA RODRIGUEZ	BIO234	21	8
MONICA SUAREZ	AST425	23	9
OSCAR MOLINA	FIS213	30	7
JULIO CORONA	MAT123	22	7
JULIETA BRAVO	MED376	18	9
TERESA LUNA	CYA436	21	6

L I B R O:

CVE_TEXTO	TITULO	PRECIO	AREA
MAT123	ALGEBRA SUPERIOR	12000	MATEMATICAS
AST425	ASTRONOMIA BASICA	25500	ASTRONOMIA
BIO234	BIOLOGIA CELULAR	40000	BIOLOGIA
FIS213	FUNDAMENTOS DE FISICA	25000	FISICA
CDI376	CALCULO DIFERENCIAL E INT.	15500	MATEMATICAS
CYA436	IMPUESTOS	30000	CONTADURIA
MED231	NEUROLOGIA	70000	MEDICINA

Si queremos los nombres de todos los lectores con menos de 20 años de edad y con un nivel de ingresos tipo 8, la consulta sería:

```
V_LEC.NOMBRE [ TAL_QUE V_LEC.EDAD < 20
               AND V_LEC.NIVEL_INGRESO > 8 ]
```

Si deseamos conocer los títulos y el precio de todos los libros del área de Biología:

```
V_LIB.TITULO,V_LIB.PRECIO [ TAL_QUE V_LIB.AREA = "BIOLOGIA" ]
```

Si deseamos conocer el nombre de todos los lectores y el título del libro que están leyendo:

```
V_LEC.NOMBRE,V_LIB.TITULO [ TAL_QUE
                            V_LEC.CVE_TEXTO = V_LIB.CVE_TEXTO ]
```

Si deseamos conocer el nombre de todos los lectores de un libro del área de Biología.

```
V_LEC.NOMBRE [ TAL_QUE
               ∃ V_LIB ( V_LIB.CVE_TEXTO = V_LEC.CVE_TEXTO
                     AND V_LIB.AREA = "BIOLOGIA" ) ]
```

Al observar la definición del cálculo relacional de eneadas notamos que se trata de una versión restringida de la lógica de predicados de primer orden, pensando específicamente en las bases de datos relacionales.

El cálculo relacional de eneadas con algunas restricciones es equivalente en su poder de expresión al algebra relacional, es decir, por cada expresión en el cálculo relacional de eneadas existe una expresión equivalente en el algebra relacional

Existe una segunda forma del cálculo relacional llamado cálculo relacional de dominios, el cual difiere del cálculo de eneadas en que sus variables toman sus valores de dominios en lugar de relaciones. Esta forma del cálculo relacional es equivalente también al algebra relacional.

3.5 RELACION CON LA LOGICA DE PREDICADOS DE PRIMER ORDEN.

Una base de datos relacional consiste en un conjunto de relaciones n-arias, junto con un conjunto de operaciones que permite la generación de nuevas relaciones a partir de las originales.

Imaginemos a una base de datos como un conjunto de cláusulas en la Lógica de predicados de primer orden, asignando a cada eneada un predicado que tendrá como nombre el de la relación al que pertenece, y como términos a los valores de sus atributos correspondientes.

Supongamos que la mayoría de las cláusulas tienen la forma p(a1,a2,...,an), es decir, son predicados (conocidos también como átomos o literales) y agrupémoslos en un conjunto de hechos o afirmaciones. Supongamos también que el número de predicados distintos es muy pequeño, comparado con el número de cláusulas.

Agrupemos por otro lado, a los mecanismos de restricción (las reglas de integridad de entidades y referenciales, los axiomas contenidos en la base de datos), en un conjunto de reglas.

```
carrera("BIO","Biólogo").

carrera("FIS","Físico").

carrera("MAT","Matemático").

carrera("ACT","Actuarío").
```

Cuando deseamos dar de alta un alumno, algunas de las restricciones podrían ser que; la clave del alumno sea un número mayor que cero y que la carrera sea una registrada en la base de datos. Estas restricciónes serían especificadas por medio de la regla siguiente:

```
alta_alumno(C_alu,N_alu,C_car):-C_alu > 0 ,
                                carrera(C_car,_),
                                inserta_alumno(C_alu,N_alu,C_car).
```

Con la cual se asegura que el alumno será registrado, utilizando el predicado inserta_alumno, siempre que la clave del alumno sea un número mayor que cero, y que la carrera sea válida.

El conjunto de este tipo de reglas representarían la intensión de la base de datos.

Seguramente ya habremos notado que: El conjunto de cláusulas de la lógica de primer orden del que tratamos no es otro sino el de las cláusulas de Horn, y que la extensión de la base de datos corresponde a los hechos mientras que la intensión corresponde a las reglas. Mas aún, uno puede escoger la manera de interpretar la ausencia de una eneada de una relación base como un enunciado cuyo valor de verdad será desconocido o falso. Si optamos por el valor desconocido hablamos de una interpretación abierta, mientras que si optamos por el valor falso tenemos una interpretación cerrada. En nuestro caso escogeremos la interpretación cerrada, ya que es congruente con el concepto de negación por falla que utiliza Prolog.

Hasta este momento hemos visto que las cláusulas de Horn son suficientes para cubrir las funciones que ofrecen los mecanismos estructurales de una base de datos relacional. En el siguiente capítulo veremos que las cláusulas de Horn también nos permiten implementar de una manera muy sencilla un lenguaje de manipulación de datos tan poderoso como el Algebra Relacional, cubriendo con esto las características que se le piden a un manejador de bases de datos para ser considerado en el modelo relacional.

CAPITULO 4

DESARROLLO DE PROGRAMAS EN PROLOG PARA LA DEFINICION, MANIPULACION E INTERROGACION DE BASES DE DATOS RELACIONALES.

En este capítulo se describe la forma de construir un manejador de bases de datos relacional utilizando el lenguaje de programación lógica PROLOG.

El manejador tendrá como objetivo permitir la definición de la estructura de la base de datos, así como el almacenamiento y recuperación de los datos de éstas estructuras.

Se han clasificado las operaciones que se pueden efectuar en la base de datos, en las que se utilizan para definición, manipulación y consulta, y son implementadas a través de comandos que permiten la comunicación entre el usuario y la base de datos; Estos comandos son similares a los del lenguaje de manipulación de datos SQL (Structured Query Languaje), el cual es uno de los lenguajes mas estándares para bases de datos relacionales.

4.1 D E F I N I C I O N.

En este inciso se describen los mecanismos de estructuración, con los cuales podemos definir relaciones y sus atributos.

Las cláusulas que representan las definiciones de relaciones y atributos, se almacenarán en una sección de la base de datos llamada "Diccionario de Datos" y que utilizaremos para asegurar que las operaciones efectuadas sean acordes con la definición de las relaciones que intervienen en la operación.

El diccionario de datos estará conformado por cláusulas cuya forma se describe a continuación.

Para la definición de relaciones :

```
tabla(relacion1,(atributo11,...,atributo1n)).
tabla(relacion2,(atributo21,...,atributo2m)).
   .
   .
   .
```

Para la definición de atributos :

```
atributo(atributo1,tipo).
atributo(atributo2,tipo).
   .
   .
   .
```

CREAR_RELACION.-

Es el procedimiento que permite definir relaciones, el cual verifica que la relación que se deséa definir no exista, revisa que sus atributos hayan sido definidos previamente, y una vez satisfechas estas dos submetas, graba una cláusula correspondiente a la definición de la relación en el diccionario de datos.

```
crear_relacion(Atributos,Relacion) :-
        not(existe_relacion(Relacion)),
        revisa_atributos(Atributos),
        assertz(tabla(Relacion,Atributos)).

existe_relacion(R) :- tabla(R).

revisa_atributos([]).
revisa_atributos([A|L]) :- atributo(A,_),
        revisa_atributos([L]).
```

Los procedimientos invocados son muy sencillos, pues el primero consiste en verificar la existencia de una cláusula "tabla" en el diccionario de datos, y la segunda, en revisar que cada uno de los atributos de la relación ya hayan sido definidos. Para grabar la definición de la relación hacemos uso del predicado assertz, un predicado predefinido de PROLOG, el cual graba una cláusula en la base de datos.

CREAR_ATRIBUTO.-

Para crear atributos utilizaremos el procedimiento siguiente, el cual verifica que el atributo no exista, revisa que su tipo sea un tipo válido, y una vez satisfechas estas dos submetas grava la definición del atributo en el diccionario de datos.

```
crear_atributo(Atributo,Tipo) :-
        not(existe_atributo(Atributo)),
        es_tipo(Tipo),
        assertz(atributo(Atributo,Tipo)).

existe_atributo(T) :- atributo(T,_).

es_tipo(T) :- tipo(T).

tipo(cadena).
tipo(entero).
```

El primer procedimiento invocado verifica la existencia de una cláusula "atributo" en el diccionario de datos, y el segundo, revisa que el tipo del atributo sea válido: cadena o entero, que por lo pronto son los dos tipos posibles. Para terminar graba la definición del atributo.

4.2 MANIPULACION.

Una vez creadas las relaciones de una base de datos, hacemos uso de los procedimientos que a continuación se describen para almacenar y actualizar la información contenida en esta.

INSERTAR_EN.-

Es el procedimiento que permite insertar eneadas (objetos) en una relación. Para esto, revisa que la relación exista y que el tipo de los datos que se desean insertar corresponda a la definición de la relación. Vigila también que la eneada no exista en la relación.

```
insertar_en(Relacion,Eneada) :-
        existe_tabla(Relacion,Atributos),
        revisa_tamanos(Eneada,Atributos),
        revisa_tipos(Eneada,Atributos),
        Objeto =.. [Relacion|Eneada],
        not(existe_eneada(Objeto)),
        assertz(Objeto).
```

Como vimos en el capítulo anterior, los registros de toda relación de la base de datos serán cláusulas que tendrán el formato anterior, el cual consiste del nombre de la relación como predicado y como términos a los valores de sus atributos correspondientes.

CAMBIAR.

Un manejador de bases de datos debe permitir al usuario hacer actualizaciones de registros que ya se encuentran en alguna relación. Este es el propósito del procedimiento cambiar, el cual verifica que la relación que se deséa actualizar exista, que el atributo cuyo valor cambiará forma parte de la definición de la relación, y que el dato que se asignará al atributo corresponda con su tipo.

```
cambiar(Atr,Rel,Op1,Operador,Op2,Valor) :-
          existe_tabla(Rel Esq),
          es_miembro(Atr,esq),
          atributo(Atr,Tipo),
          compara_tipo(Val,Tipo),
          lee_relacion(Rel,Reg),
          busca_operando(Esq,Reg,Op1,Opi),
          busca_operando(Esq,Reg,Op2,Opd),
          compare(Operador,Opi,Opd),
          construye_registro(Reg,Esq,Atr,Valor,Registro),
          Reg_actual =.. [Rel|Reg],
          retract(Reg_actual),
          Reg_nuevo =.. [Rel|Reg],
          assertz(Reg_nuevo),
          fail.

es_miembro(Atr,Esq) :- miembro(Atr,Esq).

miembro(M,[M|_]).
miembro(M,[_|T]) :- miembro(M,T).

compara_tipo :- integer(E), Tipo = entero. !.
compara_tipo :- atom(E), Tipo = cadena. !.

lee_relacion(Rel,Reg) :- tabla(Rel,Atr),
          length(Atr,Numatr),
          functor(Estruc,Rel,Numatr),
          call(Estruc),
          Estruc =.. [Rel|Reg].

busca_operando(Esq,Lista,Op,Nop) :- miembro(Op,Esq),
          posicion(Op,Esq,N,1),
          elemento(N,Lista,Nop,1),!.
busca_operando(Esq,Lisat,Op,OP) :- !.
```

```
posicion(Op,[Op|_],M,M).
posicion(Op,[H|T],N,M) :- Ind is M+1,
          posicion(Op,T,N,Ind).

elemento(N,[H|T],N,M).
elemento(N,[H|T],V,M) :- Ind is M+1,
          elemento(N,T,V,Ind).


construye_registro(Lista,Esq,Atr,Val,Reg) :-
          posicion(Atr,Esq,N,1),
          Estructura =.. [relacion|Lista],
          argrep(Estructura,N,Val,Nueva_estructura),
          Nueva_estructura =.. [relacion|Reg], !.
```

y consiste en :

a) Verificar en el diccionario de datos la existencia de la relación en la que se desea cambiar el valor de uno de sus atributos, recuperando a la vez la definición de esta. (Submeta existe_tabla).

b) Revisar si el atributo que se desea cambiar, forma parte de la definición de la relación. (Submeta es_miembro)

c) Se recupera el tipo del atributo y se compara con el tipo del valor que se le desea asignar. (Submetas **atributo** y **compara_tipo**).

d) Lee un registro de la relación. (Submeta lee_relacion). Para esto utilizamos los predicados de Prolog functor, call y **univ**, los cuales se explican a continuación.

functor(Estruc,Nombre,Numarg) .- Dada una estructura regresa el nombre de la estructura y su número de argumentos; o como se utiliza en este caso, dados el nombre de una estructura y el número de argumentos que posee, nos regresa una estructura con el nombre dado y como argumento un número de variables igual al número de argumentos dado.

call(Predicado) .- Trata de resolver la meta que se la pasa como argumento, y lo hace con éxito si la meta sucede.

Estructura =.. Lista .- Nos permite manejar un término ya sea como una lista o como una estructura. Este predicado es conocido com Univ.

El procedimiento lee_relacion consiste entonces en :

- Recupera del diccionario de datos la definición de la relación y cuenta el número de atributos que tiene.

- Con el número de atributos y con el nombre de la relación genera una estructura utilizando el predicado functor.

```
borrar_de(Relacion,Op1,Operador,Op2) :-
          existe_tabla(Relacion,Esq),
          lee_relacion(Relacion,Lista),
          busca_operando(Esq,Lista,Op1,Opi),
          busca_operando(Esq,Lista,Op2,Opd),
          compare(Operador,Opi,Opd),
          Registro =.. [Relacion|Lista],
          retract(Registro),
          fail.
```

y consiste en :

a) Verificar en el diccionario de datos la existencia de la relación, recuperando a la vez la definición de esta. (Submeta existe_tabla).

b) Lee un registro de la relación utilizando el predicado lee_relacion, el cual ya fué descrito con anterioridad.

c) Busca en el registro leído el valor de los operandos y los asigna a las variables Opi y Opd. Para esto utiliza el predicado busca_operando el cual ya fué descrito.

d) Compara los operandos utilizando el predicado compare.

e) Una vez satisfechas las submetas anteriores se procede a construir la cláusula que se eliminará de la base de datos. Para esto utilizamos el predicadoo "Univ".

f) Borramos el registro de la base de datos utilizando el predicado retract.

g) Por último, el predicado es obligado a fallar a través del predicado predefinido fail. Esto tiene como propósito provocar el backtrack, y así llegar a leer otro registro que será procesado de la misma forma. Cuando ya no es posible leer más registros, todos los elementos de la relación que cumplen con la condición especificada han sido dados de baja.

Se han programado otros comandos que, aunque permiten la manipulación de datos, no son tan relevantes como los que hemos mostrado. Estos comandos cubren las funciones siguientes.

- Cargar o Salvar una base de datos.

    cargar(archivo).

    salvar(archivo).

- Mostrar la descripción de una relación o listar las relaciones existentes en la base de datos.

```
relacion(nombre).
relaciones.
```

- Mostrar la descripción de los atributos existentes.

```
atributos.
```

- Borrar una relación de la base de datos.

```
borrar_relacion(nombre).
```

No se dará una descripción detallada de estos procedimientos, pero es posible consultarla en al anexo-A donde se presenta un listado completo de el programa en cuestión.

4.3 C O N S U L T A.

En un lenguaje de manipulación de datos, la información es recuperada de la base de datos a través del comando seleccionar, en el cual; se da el nombre de la relación(es) que se desea consultar, se describen las columnas que se desean como respuesta, y se establece una condición que deben cumplir los registros seleccionados.

SELECCIONAR.

Este comando es inplementado en Prolog con las siguientes cláusulas, las cuales verifican que la relación en cuestión exista, que los atributos que se desea consultar pertenezcan a la definición de ésta, y por último, procesan secuencialmente la relación seleccionando los registros que cumplen la condición dada.

```
seleccionar(Atr,Rel,Op1,Operador,Op2) :-
          existe_tabla(Rel,Esq),
          atributos_validos(Esq,Atr),
          lee_relacion(Rel,Lista),
          busca_operando(Esq,Lista,Op1,Opi),
          busca_operando(Esq,Lista,Op2,Opd),
          compare(Operador,Opi,Opd),
          selecciona_campos(Lista,Esq,Atr,Consulta,[]),
          write(Consulta),
          fail.


atributos_validos(A,[*|[]]) :- !.
atributos_validos(A,[]) :- !.
atributos_validos(A,[H|T] :- miembro(H,A), !,
          atributos_validos(A,T).
```

```
selecciona_campos(Lista,Esq,[*],Lista,_) :- !.
selecciona_campos(Lista,Esq,[],L,L).
selecciona_campos(Lista,Esq,[A|R],Consulta,L) :-
        posicion(A,Esq,N,1),
        elemento(N,Lista,Ele,1),
        append(L,[Ele],Nueva),
        selecciona_campos(Lista,Esq,R,Consulta,Nueva).
```

que conforman el procedimiento seleccionar el cual consiste en:

a) Verifica en el diccionario de datos la existencia de la relación, recuperando a la vez la definición de esta. (Submeta existe_tabla).

b) Verifica que los atributos que se desean recuperar formen parte de la definición de la relación. (Submeta atributos_validos).

c) Lee un registro de la relación utilizando el predicado lee_relacion.

d) Busca en el registro leído el valor de los operandos y los asigna a las variables Opi y Opd. (Submeta busca_operando).

e) Compara los operandos utilizando el predicado compare.

f) Selecciona de el registro leído los valores de los atributos que se desean recuperar. (Submeta selecciona_campos). En éste procedimiento se construye la lista Consulta, la cual está formada por la lista de valores asociados a los atributos requeridos, o por el registro completo en el caso de que la lista de atributos sea el caracter "*".

g) Despliega la lista resultante en el paso anterior, la cual, es la consulta deseada.

h) Por último, el predicado es obligado a fallar a través del predicado predefinido fail. Esto tiene como propósito provocar el backtrack, y así llegar a leer otro registro que será procesado de la misma forma. Cuando ya no es posible leer más registros, todos los elementos de la relación que cumplen con la condición especificada han sido desplegados.

UNION.

Dadas dos relaciones, es posible construir una nueva relación la cual contiene a los registros de cada una de las relaciones. Esta relación es la unión.

Existe una diferencia entre esta unión y la que se utiliza en conjuntos y consiste en que ; sólo pueden ser operadas relaciones que tienen el mismo número de atributos y que sus tipos correspondientes sean los mismos. Esto es consistente con la forma de definir una relación, pues si pudiesen ser operadas cualquier par de ellas podrían dar como resultado una relación donde no se podría tener una única definición de sus registros.

A las relaciones que cumplen la condición anterior se los conoce como "Unión-compatibles".

El procedimiento de unión es el siguiente:

```
union(A,B) :- compatibles(A,B),
          a_union_b(A,B,Lista),
          write(Lista),
          fail.

a_union_b(A,B,Lista) :- lee_relacion(A,Lista) ;
          ( lee_relacion(B,Lista) ,
            not(lee_relacion(A,Lista))).

compatibles(A,B) :- tabla(A,La), tabla(B,Lb),
          revisa_tamanos(La,Lb),
          compara_atributos(La,Lb), !.


compara_atributos([],[]) :- !.
compara_atributos([A|Ra],[B|Rb]) :-
          atributo(A,Ta), atributo(B,Tb),
          compara_tipos(Ta,Tb),
          compara_atributos(Ra,Rb).

compara_tipos(Ta,Tb) :- Ta = Tb, !.
```

y consiste en,

a) Verificar que las relaciones sean compatibles bajo la unión (submeta compatibles). Para esto lee la descripción de las relaciones del diccionario de datos, verifica que tengan el mismo número de atributos y que sus tipos correspondientes sean iguales (submetas revisa_tamanos y compara_atributos).

b) Lee uno a uno los registro de la relación A y cuando ha leído todos, lee la relación B. Este paso se consigue utilizando el operador OR denotado por ";".

Dado que en un conjunto no es posible tener elementos repetidos, cuando se leen registros de la segunda relación se les añade la condición que no estén contenidos en la relación A.

c) Se despliega el registro leído.

d) Se provoca el backtrack, con el predicado fail, para así poder leer el resto de registros.

En los siguientes comandos es necesario, para poder aplicarlos, que las relaciones que son operadas cumplan con la propiedad de ser compatibles bajo la unión.

INTERSECCION.

Tiene también una descripción muy simple

```
interseccion(A,B) :- compatibles(A,B),
          a_interseccion_b(A,B,Lista),
          write(Lista),
          fail.

a_interseccion_b(A,B,Lista) :- lee_relacion(A.Lista),
          lee_relacion(B,Lista).
```

y consiste en verificar que el registro leído exista en las dos relaciones.

DIFERENCIA.

Su descripción es :

```
diferencia(A,B) :- compatibles(A,B),
          a_diferencia_b(A,B,Lista),
          write(Lista),
          fail.

a_diferencia_b(A,B,Lista) :- lee_relacion(A,Lista),
          not(lee_relacion(B,Lista)).
```

y regresa como resultado los registros que están contenidos en la relación A pero no en B.

```
a_division_b(_,_,_,_) :- seleccionar([*],temp,1,=,1).

a_division_b(A,B,Nea,Neb) :- borrar_registros(temp),
          Tab =.. [Tabla|[temp,_]],
          retract(Tab).

verifica_b(A,B,Slist) :-
          lee_relacion(B,Lb),
          append(Slist,Lb,Lista),
          Eneada =.. [A|Lista],
          llamar(Eneada,Slist).

llamar(Eneada,_) :- call(Eneada), !.

llamar(Eneada,Slist) :- baja_div(Slist), !.
```

y tiene la siguiente descripción:

a) Revisa que las relaciones A y B sean divisibles utilizando el predicado divisibles que se explica a continuación.

- Verifica que las relaciones existan, recupera sus descripciones y calcula sus tamaños. Predicados existe_tabla y length.

- Verifica que la descripción de la relación B (divisor) esté contenida en la relación A (dividendo). Predicados sublista y compara_atributos.

- Construye la descripción de una relación temporal, utilizando la sublista de la descripción de A formada por los campos que no están en B. La sublista construida será la descripción de la relación donde se almacenará el resultado de la división.

- Crea una entrada en el diccionario de datos con la descripción de la relación temporal. Predicado assertz.

b) Lee un registro de A ,construye una lista con las campos que son únicamente de A ,y si no existe en la relación temporal, la da de alta.

c) Para cada registro de A procesa la relación B, concatenando la lista de campos que son únicamente de A con un registro de B. De esta forma es generado un registro que tiene la misma estructura que los de la relación A.

d) Revisa que el registro generado en el punto anterior esté contenido en A. Si ésto sucede se pasa a leer otro registro de B y se le aplica el mismo procedimiento. Si el registro no está en A, se elimina de la relación temporal la entrada correspondiente y se pasa a leer otro registro de A.

GEN_FCAN.

Este procedimiento recibe como argumento de entrada la lista generada por gen_lista, y la transforma en una lista donde se han omitido los elementos irrelevantes para la construcción de una consulta en forma de cláusulas de Prolog.

Como un ejemplo supongamos que el usuario escribe el comando:

"borrar_de Socio donde cuenta = 123"

El procedimiento gen_lista al procesar esta consulta regresará la lista :

[borrar_de,socio,donde,cuenta,=,123]

El procedimiento gen_fcan, el cual consta de varias cláusulas, unificará la llamada

gen_fcan(Lista,Nueva_lista)

con la cláusula

```
gen_fcan( ['borrar_de'|[R|['donde'|[Op1|[Opd|[Op2|[]]]]]]] ,
          ['borrar_de',R,Op1,Opd,Op2] ).
```

generando la meta

```
gen_fcan(['borrar_de'|[socio|['donde'|[cuenta|[=|[123|[]]]]]]] ,
        ['borrar_de',socio,cuenta,=,123] ).
```

donde su segundo argumento es una lista formada únicamente por los elementos de la lista original que pueden ser utilizados para construir la consulta.

De esta forma se asegura que la consulta tenga un formato que pueda ser transformado y donde han sido eliminados los elementos de la lista original que son irrelevantes para la consulta (en este caso la palabra 'donde').

Al unificar la variable Nueva_lista con el segundo argumento de la cláusula le es asignada la lista que tiene un formato fijo,

Nueva_lista = [borrar_de,socio,cuenta,=,123]

la cual podemos pasar como argumento a la rutina que ejecuta el comando.

EJECUTA.

En esta parte del programa se llama a los procedimientos de definición, manipulación e interrogación . Esto se logra con el comando ejecuta(Nueva_lista) , cuyo único argumento es el resultado del, procedimiento gen_fcan.

```
ejecuta(Nueva_lista) :- Predicado =.. Nueva_lista
                        call(Predicado).
```

El procedimiento construye una meta, haciendo uso del predicado Univ., la cual será ejecutada a través del predicado call.

Retomando el ejemplo anterior, el procedimiento ejecuta construye la variable:

```
Predicado = borrar_de(socio,cuenta,=,123)
```

que al ser ejecutado por call

```
call(borrar_de(socio,cuenta,=,123))
```

se unificará con el procedimiento borrar_de que ya hemos descrito en el inciso 4.2 .

BD (Meta Inicial).

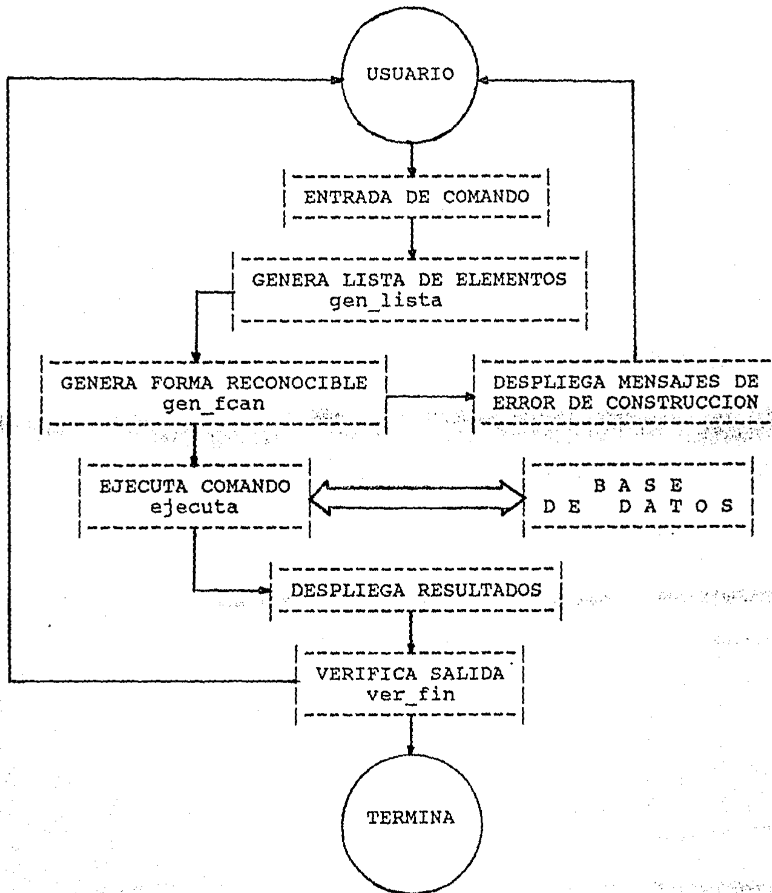
Este predicado inicia la ejecución de la aplicación, y genera un proceso cíclico que pide al usuario un comando.

```
bd :- repeat,
      lee_comando(Comando),
      gen_lista(Comando,[],Lista)
      gen_fcan(Lista,Consulta),
      ejecuta(Consulta),
      ver_fin(Consulta), !.
```

El predicado predefinido repeat hace que los predicados que le siguen sean ejecutados sucesivamente. Este ciclo es detenido cuando el predicado ver_fin es resuelto con éxito, enseguida,se resuelve el corte (!) con éxito y por lo tanto ya no es posible volver a ejecutar el ciclo, terminando así la ejecución del programa.

El predicado ver_fin sucede cuando el usuario da el comando terminar.

A continuación se muestra un diagrama del proceso que se realiza cuando un usuario hace una consulta, y que resume los predicados que fueron descritos anteriormente:



4.5 EJEMPLO DE OPERACION.

Mostraremos a continuación un ejemplo de la operación del programa utilizando algunos de los comandos descritos anteriormente.

Sintaxis de Comandos :

CREAR_ATRIBUTO <Atributo> <Tipo>

CREAR_RELACION <Relación> con <Atributos>

INSERTAR_EN <Relación> <Registro>

RELACIONES

RELACION <Relación>

SELECCIONAR <Lista_Atributos> de <Relación> [donde <Condición>]

CAMBIAR <Atributo> = <Valor> en <Relación> donde <Condición>

UNION_THETA <Relación> <Relación> donde <Condición>

BORRAR_RELACION <Relación>

BORRAR_DE <Relación> donde <Condición>

SALVAR <Archivo>

CARGAR <Archivo>

UNION <Relación> <Relación>

INTERSECCION <Relación> <Relación>

DIFERENCIA <Relación> <Relación>

PRODUCTO <Relación> <Relación>

TERMINAR

donde:

Relación = Nombre de relación.

Atributo = Nombre de atributo.

Tipo = Nombre de un tipo de dato.

Registro = Lista de valores.

Valor = Atomo, elemento de un dominio.

Archivo = Nombre de archivo que contiene a una Base de Datos.

Condición = tiene la forma Operando Operador Operando.

Operando = Puede ser un Atributo o un Valor

Operador = ">" , "<" , "="

Supongamos que deseamos crear una base de datos que contiene información referente a una compañía que se dedica a la compra y venta de materiales de construcción.

La base de datos podría constar de las relaciones siguientes:

RELACION — **A T R I B U T O S**

CLIENTE :

Nombre	Telefono	Vend
'Manuel Velazquez'	5909269	v1
'Maria Elena Trejo'	2868133	v7
'Patricia Luna'	5623483	v3
'Oscar Velazquez'	3984671	v7

VENDEDOR :

Nombre	Vend	Comision
'Javier Sanchez'	v1	28728
'Miguel Garcia'	v7	32453
'Mario Laguna'	v3	21872

PRODUCTO :

Prod	Compra	Venta
'cemento'	150	180
'varilla'	280	320
'arena'	28	27
'piedra'	50	55

Iniciamos el ejemplo dando la meta bd. con lo cual el programa nos pide un comando.

```
<<< Comando >>> :
```

A continuación creamos los atributos que necesitamos.

```
<<< Comando >>> : crear_atributo nombre cadena
<<< Comando >>> : crear_atributo telefono entero
<<< Comando >>> : crear_atributo vendedor cadena
<<< Comando >>> : crear_atributo comision entero
<<< Comando >>> : crear_atributo prod cadena
<<< Comando >>> : crear_atributo compra entero
<<< Comando >>> : crear_atributo venta entero
```

Creamos ahora las relaciones.

```
<<< Comando >>> : crear_relacion cliente con nombre
                                              telefono
                                              vend

<<< Comando >>> : crear_relacion vendedor con nombre
                                               vend
                                               comision

<<< Comando >>> : crear_relacion producto con prod
                                               compra
                                               venta
```

Insertamos los registros.

```
<<< Comando >>> : insertar_en cliente 'Manuel Velazquez'
                                      5909269
                                      v1
<<< Comando >>> : insertar_en cliente 'Maria Elena Trejo'
                                      2868133
                                      v7
<<< Comando >>> : insertar_en cliente 'Patricia Luna'
                                      5623483
                                      v3
<<< Comando >>> : insertar_en cliente 'Oscar Velazquez'
                                      3984671
                                      v7

<<< Comando >>> : insertar_en vendedor 'Javier Sanchez'
                                        v1
                                        28728
<<< Comando >>> : insertar_en vendedor 'Miguel Garcia'
                                        v7
                                        32453
<<< Comando >>> : insertar_en vendedor 'Mario Laguna'
                                        v3
                                        21872

<<< Comando >>> : insertar_en producto cemento 150 180
<<< Comando >>> : insertar_en producto varilla 280 320
<<< Comando >>> : insertar_en producto arena 28 27
<<< Comando >>> : insertar_en producto piedra 50 55
```

Procedemos a efectuar las consultas:

Queremos saber el precio de compra del cemento:

```
<<< Comando >>> : seleccionar compra de producto donde
                              prod = cemento
```

150

Queremos una lista del total de productos y su precio de venta.

```
<<< Comando >>> : seleccionar prod venta de producto

cemento 180
varilla 320
arena   27
piedra  55
```

Deseamos los nombres de los clientes del vendedor "v7"

```
<<< Comando >>> : seleccionar nombre de cliente donde vend = v7

Maria Elena Trejo
Oscar Velazquez
```

Listar la relación producto completa

```
<<< Comando >>> : seleccionar * de producto

cemento 150 180
varilla 280 320
arena    28  27
piedra   50  55
```

Queremos saber que productos tienen un precio de compra mayor al de venta

```
<<< Comando >>> : seleccionar * de producto donde
                              compra > venta

arena 28 27
```

Queremos borrar el registro de la relación producto que corresponde a la piedra.

```
<<< Comando >>> : borrar_de producto donde prod = piedra
```

Queremos cambiar el precio de venta de la arena a 77

```
<<< Comando >>> : cambiar venta = 77 en producto donde
                                                  prod = arena
```

Listar la relación producto completa

```
<<< Comando >>> : seleccionar * de producto

cemento 150 180
varilla 280 320
arena    28  77
```

Queremos respaldar la base de datos en el archivo "archbd"

```
<<< Comando >>> : salvar archbd
```

Deseamos ahora cargar la base de datos "set"

```
<<< Comando >>> : cargar set
```

Queremos saber que relaciones tiene

```
<<< Comando >>> : relaciones

rel1 [a,b]
rel2 [c]
rel3 [d]
```

Consultamos las relaciones

```
<<< Comando >>> : seleccionar * de rel1

p 1
p 2
p 3
q 1
r 1
r 3

<<< Comando >>> : seleccionar * de rel2

1
3

<<< Comando >>> : seleccionar * de rel3

1
5
9
```

Efectuamos las operaciones unión, intersección, diferencia y division.

```
<<< Comando >>> : union rel2 rel3

1
3
5
9

<<< Comando >>> : interseccion rel2 rel3

1

<<< Comando >>> : diferencia rel3 rel2

5
9

<<< Comando >>> : division rell rel2

p
r
```

Encontramos la Unión-Theta de las relaciones rell y rel2.

```
<<< Comando >>> : union_theta rell rel2 donde b = c

p 1 1
p 3 3
q 1 1
r 1 1
r 3 3
```

Terminamos la operación del programa

```
<<< Comando >>> : terminar
```

C O N C L U S I O N E S

Este trabajo es una muestra de la comunicación existente entre dos áreas de investigación que originalmente se desarrollaban en forma separada, las Bases de Datos y la Inteligencia Artificial. La parte en común de estas dos áreas es una diciplina llamada Programación Lógica la cual ha ganado atención en la tecnología de Bases de Datos Relacionales.

En éste trabajo, se desarrolló un Manejador de Bases de Datos usando PROLOG, un lenguaje de programación lógica que ha mostrado ser una herramienta que permite el desarrollo de una manera muy simple.

Los aspectos estructurales del modelo relacional permiten ver a una base de datos relacional como un conjunto de fórmulas de la Lógica de Predicados de Primer Orden, por lo que PROLOG se presenta como un lenguaje ideal para el desarrollo de bases de datos y, como es mostrado en el trabajo, para el desarrollo de manejadores de bases de datos relacionales.

Durante la fase de programación fué necesario cambiar de implementación del lenguaje. Se inició con Turbo-Prolog, pero se presentaron contratiempos ya que el lenguaje impone una estructura rígida a los programas, y carece de predicados predefinidos muy útiles como *Functor*, *Univ* y *Call*. En su lugar se eligió trabajar con Arity-Prolog, obteniendo así una mayor similitud con la programación lógica.

Se eligió desarrollar los comandos que implementan las operaciones, de una forma similar a los de SQL, por ser éste uno de los lenguajes de manipulación de datos mas estándares.

En el programa se dió mas atención a la forma en que son representados los datos, y a las operaciones que pueden ser efectuadas con ellos, que a la interfaz con el usuario (la forma de expresar las consultas y como son dadas las respuestas).

El programa como ya se mencionó, presenta limitantes en algunos predicados, pues el propósito fundamental fué el mostrar que las operaciones básicas de un manejador de base de datos

relacional pueden ser desarrolladas facilmente a traves de Prolog. Entre las restricciones mas importantes se encuentran las siguientes :

- Los atributos que definen una relación pueden ser únicamente de dos tipos: cadena y entero.

- Los operadores que intervienen en expresiones lógicas son el de igualdad, menor y mayor.

- El único caracter delimitador es el blanco.

- No se implementó la creación de llaves en una relación.

- No se permiten las consultas anidadas.

Por otro lado, el programa cumple con todas las propiedades necesarias para el manejo de relaciones, desde las operaciones básicas (unión, intersección, diferencia, producto cartesiano, división), hasta operaciones mas sofisticadas como la selección, la unión-theta, el cambio de un valor de una eneada, el alta y baja de registros. Fueron programados también los mecanismos de definición de relaciones.

En resumen, el programa tiene caracteristicas (como las operaciones entre conjuntos) que no contemplan algunas aplicaciones que se encuentran en el mercado y que se hacen llamar, o dicen contener, "bases de datos", pero es inferior en cuanto a la interfaz que estos ofrecen al usuario.

Entre las tareas que pueden complementar el programa se encuentran : La implementación de mecanismos para garantizar la integridad referencial y la aceptación de consultas anidadas.

Lo anterior está de acuerdo con las tendencias de desarrollo en el área de Bases de datos Relacionales en las que la Programación Lógica ha mostrado su utilidad, éstas tendencias son :

- El desarrollo de interfases amigables para las bases de datos, donde se contemplen técnicas de procesamiento de lenguajes formales como SQL.

- Interfases en lenguaje natural, donde los usuarios puedan expresarse en términos familiares sin tener que conocer la estructura de la información, y sin tener que aprender un lenguaje formal.

Como puede ser visto la Programación Logica a dejado atras su aplicación original, que fué la de ser una herramienta para la demostración de teoremas, y se ha convertido en un lenguaje muy útil en el desarrollo de aplicaciones en el área de las Bases de Datos Relacionales.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%% bd. ( Procedimiento Inicial ) %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

bd :-
     repeat,nl,
     lee_comando(Comando),
     gen_lista(Comando,[],Lista),
     gen_fcan(Lista,Fcan),
     ejecuta(Fcan),
     ver_fin(Fcan), ! .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% lee_comando %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lee_comando(Comando) :-
     nl,
     write('<<< Comando >>> : '),
     read(Comando).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% gen_lista %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gen_lista([H|T],Lista,L) :-
     digito(H),!,
     resto_numero(T,[H],Numero,Resto),
     append(Lista,[Numero],Nlista),
     gen_lista(Resto,Nlista,L).

gen_lista([H|T],Lista,L) :-
     letra(H,Letra),!,
     resto_nombre(T,[Letra],Nombre,Resto),
     append(Lista,[Nombre],Nlista),
     gen_lista(Resto,Nlista,L).

gen_lista([H|T],Lista,L) :-
     token(H,Token),
     append(Lista,[Token],Nlista),
     gen_lista(T,Nlista,L).

gen_lista([32|T],Lista,L) :-
     !, gen_lista(T,Lista,L).

gen_lista([],Lista,Lista).

resto_numero([H|T],Lista,Numero,Resto) :-
     digito(H),!,
     append(Lista,[H],Nlista),
     resto_numero(T,Nlista,Numero,Resto).

resto_numero([32|T],Lista,Numero,Resto) :-
     name(Numero,Lista), Resto = T, !.
```

```
resto_numero([H|T],Lista,Numero,[H|T]) :-
     token(H,_),
     name(Numero,Lista),!.

resto_numero([],Lista,Numero,[]) :-
     !, name(Numero,Lista).

resto_numero([H|T],Lista,Numero,Resto) :-
     resto_numero(T,Lista,Numero,Resto).

resto_nombre([H|T],Lista,Nombre,Resto) :-
     caracter_valido(H,Caracter),!,
     append(Lista,[Caracter],Nlista),
     resto_nombre(T,Nlista,Nombre,Resto).

resto_nombre([32|T],Lista,Nombre,Resto) :-
     name(Nombre,Lista), Resto = T, !.

resto_nombre([],Lista,Nombre,[]) :-
     !, name(Nombre,Lista).

token('=,=).
token('>,>).
token('<,<).
token('*,*).

digito(N) :- N >= '0 , N =< '9 .

letra(C,L) :-
     C >= 'A , C =< 'Z,
     L is C + 32 .

letra(L,L) :- L >= 'a , L =< 'z .

caracter_valido(C,C) :- digito(C).

caracter_valido(C,L) :- letra(C,L).

caracter_valido('_,'_).

append([],Lista,Lista).
append([H|T],L,[H|T1]) :- append(T,L,T1).

ver_fin(['terminar']) :- nl, write('<<< FIN DE SESION >>>').
ver_fin(_) :- fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% gen_fcan %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gen_fcan(['crear_atributo'|[A|[T|[]]]],['crear_atributo',A,T]) :- !.

gen_fcan(['crear_relacion'|[R|['con'|A]]],['crear_relacion',A,R]) :- !.
```

ESTA TESIS NO DEBE SALIR DE LA BIBLIOTECA

```
gen_fcan(['insertar_en'|[R|E]],['insertar_en',R,E]) :- !.

gen_fcan(['relaciones'|[]],['relaciones']) :- !.

gen_fcan(['atributos'|[]],['atributos']) :- !.

gen_fcan(['relacion'|[R|[]]],['relacion',R]) :- !.

gen_fcan(['borrar_relacion'|[R|[]]],['borrar_relacion',R]) :- !.

gen_fcan(['salvar'|[A|[]]],['salvar',A]) :- !.

gen_fcan(['cargar'|[A|[]]],['cargar',A]) :- !.

gen_fcan(['union'|[A|[B|[]]]],['union',A,B]) :- !.

gen_fcan(['interseccion'|[A|[B|[]]]],['interseccion',A,B]) :- !.

gen_fcan(['diferencia'|[A|[B|[]]]],['diferencia',A,B]) :- !.

gen_fcan(['producto'|[A|[B|[]]]],['producto',A,B]) :- !.

gen_fcan(['division'|[A|[B|[]]]],['division',A,B]) :- !.

gen_fcan(['terminar'|[]],['terminar']) :- !.

gen_fcan(['union_theta'|
        [R1|[R2|['donde'|[Op1|[Opd|[Op2|[]]]]]]]],
        ['union_theta',R1,Op1,R2,Op2,Opd]) :- !.

gen_fcan(['borrar_de'|
        [R|['donde'|[Op1|[Opd|[Op2|[]]]]]]],
        ['borrar_de',R,Op1,Opd,Op2]) :- !.

gen_fcan(['cambiar'|
        [A|['='|[V|['en'|[R|['donde'|[Op1|[Opd|[Op2|[]]]]]]]]]]],
        ['cambiar',A,R,Op1,Opd,Op2,V]) :- !.

gen_fcan(['seleccionar'|L],['seleccionar',Atributos,R,Op1,Opd,Op2]) :- !,
     divide(L,[],Atributos,Resto),!,
     asigna(Resto,R,Op1,Opd,Op2).

divide([],_,_,[]).
divide(['de'|R],A,A,R).
divide([H|T],La,A,R) :-
     append(La,[H],NLa),
     divide(T,NLa,A,R).

asigna([R|['donde'|[Op1|[Opd|[Op2|[]]]]]],R,Op1,Opd,Op2).
asigna([R|[]],R,1,=,1).

gen_fcan([_|_],[_]) :- nl, write('< Comando Invalido >').
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% ejecuta %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ejecuta(Comando) :-
     Predicado =.. Comando,
     call(Predicado).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% Comandos %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% crear_atributo (nombre,tipo) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

crear_atributo(A,T) :- not(existe_atributo(A)),
     es_tipo(T),
     assertz(atributo(A,T)),
     nl, write(A), write(' : '), write(T), !.

existe_atributo(A) :- atributo(A,_),
     nl, write('< ATRIBUTO EXISTENTE >'), !.

es_tipo(T) :- tipo(T), !.
es_tipo(T) :- nl, write('< TIPO INVALIDO >'), fail.

tipo(cadena).
tipo(entero).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% cargar (archivo) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cargar(A) :- concat(A,'.dbf',Arch),
     reconsult(Arch),
     nl, write('< ARCHIVO CARGADO : '), write(Arch), write('>'), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% salvar (archivo) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

salvar(A) :- concat(A,'.dbf',Arch),
     tell(Arch),
     listing(atributo),
     listing(tabla),
     salvabd,
     told,
     nl, write('< ARCHIVO SALVADO : '), write(Arch), write('>'), !.


salvabd :- tabla(R,_), listing(R), fail.
salvabd.
```

```
revisa_tipos([],[]):- !.
revisa_tipos([E|T],[A|R]) :- atributo(A,Tipo),
     compara_tipo(E,Tipo),
     revisa_tipos(T,R).

compara_tipo(E,Tipo) :- integer(E), Tipo = entero, !.
compara_tipo(E,Tipo) :- atom(E), Tipo = cadena, !.
compara_tipo(E,Tipo) :- nl, write('< TIPO DE DATO INVALIDO : '),
     write(Tipo), write(' : '), write(E), write(' >'),
     fail.

existe_eneada(Eneada) :- call(Eneada), nl, write('< YA EXISTE OBJETO >'),

insertar_en(_,_) :- nl, write('< ALTA RECHAZADA >').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% relacion(nombre) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

relacion(R) :- tabla(R,A),
     nl, write('RELACION = '), write(R),
     lista_atributos(A), !.

lista_atributos([]) :- !.
lista_atributos([A|Resto]) :- atributo(A,Tipo),
     nl, write(A), write(' ('), write(Tipo), write(')'),
     lista_atributos(Resto).

relacion(_) :- nl, write('< RELACION INEXISTENTE >').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% borrar_relacion (relacion) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

borrar_relacion(R) :- tabla(R,A),
     borrar_registros(R),
     Tab =.. [tabla|[R,_]],
     retract(Tab),
     nl, write('< RELACION DADA DE BAJA >'), !.

borrar_registros(R) :- Reg =.. [R|_],
     retract(Reg), fail.
borrar_registros(_).

borrar_relacion(_) :- nl, write('< RELACION INEXISTENTE >').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% union(relacion,relacion) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

union(A,B) :- compatibles(A,B),
     a_union_b(A,B,Lista),
     nl,write(Lista), fail.
```

```
a_union_b(A,B,Lista) :- lee_relacion(A,Lista) ;
     ( lee_relacion(B,Lista), not(lee_relacion(A,Lista)) ).

compatibles(A,B) :- existe_tabla(A,La), existe_tabla(B,Lb),
     revisa_tamanos(La,Lb),
     compara_atributos(La,Lb), !.

compara_atributos([],[]):- !.
compara_atributos([A|Ra],[B|Rb]) :-
     atributo(A,Ta), atributo(B,Tb),
     compara_tipos(Ta,Tb),
     compara_atributos(Ra,Rb).

compara_tipos(Ta,Tb) :- Ta = Tb, !.
compara_tipos(_,_) :-  nl, write('< TIPOS NO CORRESPONDEN >'), fail.

lee_relacion(Rel,Reg) :- tabla(Rel,Atr),
     length(Atr,Numatr),
     functor(Estruc,Rel,Numatr),
     call(Estruc),
     Estruc =.. [Rel|Reg].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% interseccion(relacion,relacion) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

interseccion(A,B) :- compatibles(A,B),
     a_interseccion_b(A,B,Lista),
     nl,write(Lista), fail.

a_interseccion_b(A,B,Lista) :- lee_relacion(A,Lista) , lee_relacion(B,Lis

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% diferencia(relacion,relacion) %%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

diferencia(A,B) :- compatibles(A,B),
     a_diferencia_b(A,B,Lista),
     nl,write(Lista), fail.

a_diferencia_b(A,B,Lista) :- lee_relacion(A,Lista) , not(lee_relaoion(B,L


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% producto(relacion,relacion) %%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

producto(A,B) :- a_producto_b(A,B,Lista),
     nl,write(Lista), fail.

a_producto_b(A,B,Lista) :- lee_relacion(A,La),
                           lee_relacion(B,Lb),
                           append(La,Lb,Lista).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% borrar_de(Relacion,Operador,Operando,Operador) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

borrar_de(Rel,Op1,Operador,Op2) :- existe_tabla(Rel,Esq),
     lee_relacion(Rel,Lista),
     busca_operando(Esq,Lista,Op1,Opi),
     busca_operando(Esq,Lista,Op2,Opd),
     compare(Operador,Opi,Opd),
     Eneada =.. [Rel|Lista],
     retract(Eneada),
     nl, write(Eneada), fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% cambiar(Atributo,Relacion,Operando,Operador,Operando,Valor) %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cambiar(Atr,Rel,Op1,Operador,Op2,Val) :- existe_tabla(Rel,Esq),
     es_miembro(Atr,Esq),
     atributo(Atr,Tipo), compara_tipo(Val,Tipo),
     lee_relacion(Rel,Lista),
     busca_operando(Esq,Lista,Op1,Opi),
     busca_operando(Esq,Lista,Op2,Opd),
     compare(Operador,Opi,Opd),
     construye_registro(Lista,Esq,Atr,Val,Reg),
     Reg_actual =.. [Rel|Lista], retract(Reg_actual),
     Reg_nuevo =.. [Rel|Reg], assertz(Reg_nuevo),
     nl, write(Reg_nuevo),
     fail.

es_miembro(Atr,Esq) :- miembro(Atr,Esq), !.
es_miembro(Atr,_):- nl, write('< ATRIBUTO INVALIDO : '),
     write(Atr), write('>'), !, fail.

construye_registro(Lista,Esq,Atr,Val,Reg) :- posicion(Atr,Esq,N,1),
     Estructura =.. [relacion|Lista],
     argrep(Estructura,N,Val,Nueva_estructura),
     Nueva_estructura =.. [relacion|Reg], !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% union_theta(Relacion1,Operando1,Relacion2,Operando2,Operador) %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

union_theta(Rel1,Op1,Rel2,Op2,Operador) :- existe_tabla(Rel1,Esq1),
     existe_tabla(Rel2,Esq2),
     es_miembro(Op1,Esq1),
     es_miembro(Op2,Esq2),
     lee_relacion(Rel1,Lista1),
     lee_relacion(Rel2,Lista2),
     busca_operando(Esq1,Lista1,Op1,Opi),
     busca_operando(Esq2,Lista2,Op2,Opd),
     compare(Operador,Opi,Opd),
     append(Lista1,Lista2,Lista),
     nl, write(Lista), fail.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% division(relacion,relacion) %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

division(A,B) :- divisibles(A,B,Nea,Neb),
     a_division_b(A,B,Nea,Neb).

divisibles(A,B,Nea,Neb) :- existe_tabla(A,Esqa), existe_tabla(B,Esqb),
     length(Esqa,Nea), length(Esqb,Neb),
     Nea > Neb,
     Pini is ( Nea - Neb ) + 1 ,
     sublista(Pini,Nea,Esqa,Subesqa),
     compara_atributos(Subesqa,Esqb),
     Pfin is ( Nea - Neb ),
     sublista(1,Pfin,Esqa,Desc),
     assertz(tabla(temp,Desc)), !.

divisibles(_,_,_,_) :-  nl, write('<RELACIONES INDIVISIBLES >'), !, fail.

a_division_b(A,B,Nea,Neb) :-
     lee_relacion(A,La),
     Pfin is ( Nea - Neb ) ,
     sublista(1,Pfin,La,Slist),
     alta_div(Slist),
     verifica_b(A,B,Slist) ,
     fail.

a_division_b(_,_,_,_) :- seleccionar([*],temp,1,=,1).
a_division_b(_,_,_,_) :- borrar_registros(temp),
     Tab =.. [tabla|[temp,_]],
     retract(Tab).

verifica_b(A,B,Slist) :-
     lee_relacion(B,Lb),
     append(Slist,Lb,Lista),
     Eneada =.. [A|Lista],
     llamar(Eneada,Slist).

llamar(Eneada,_) :-  call(Eneada), !.
llamar(Eneada,Slist) :- baja_div(Slist), !.

alta_div(Slist) :-
     Eneada =..[temp|Slist],
     not(call(Eneada)),
     assertz(Eneada), !.
alta_div(_).

baja_div(Slist) :-
     Eneada =.. [temp|Slist],
     call(Eneada),
     retract(Eneada), !.
baja_div(_).
```

```
sublista(Ini,Fin,Entra,Sale) :-
     sub1(Entra,Ini,Temp),
     X is Fin - Ini + 1,
     sub2(Temp,X,Sale).

sub1(Lista,1,Lista) :- !.
sub1([_|Lista],N,Temp) :-
     N > 0,
     NN is N - 1,
     sub1(Lista,NN,Temp).

sub2(_,0,[]) :- !.
sub2([H|Lista],N,[H|Temp]) :-
     N > 0,
     NN is N - 1,
     sub2(Lista,NN,Temp).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% fin del programa %%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

ANEXO B

B I B L I O G R A F I A

[1] J. W Lloyd
Foundations of Logic Programming
Springer-Verlag, 1984.

[2] Cristopher John Hogger
Introduction to Logic Programming
Academic Press, 1984.

[3] Carl Townsend
Advanced Techniques in Turbo Prolog
Sibex, 1987.

[4] Claudia Marcus
Prolog Programming
Addison-Wesley, 1986.

[5] C. J. Date
An Introduction to Database Systems
Addison-Wesley, 1981.

[6] Clocksin, W.F. and Mellish, C.S.
Programming in Prolog
Springer-Verlag, 1984.

[7] Feliks Klusniak, Stanislaw Szpakowicz
Prolog for Programmers
Academic Press, 1985.

[8] Jacques Cohen
Describing Prolog by its interpretation and compilation
Cominicaciones ACM, 1985.

[9] John Malpas
Programming in Logic
Dr. Dobb's Journal, 1985

[10] Dean A. Schlobohm
Introduction to Prolog
Robotics Age, 1984

[11] David E. Cortesi
A Tour of Prolog
Dr. Dobb's Journal, 1985