UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

NORMALIZACION DE RELACIONES CON TECNICAS DE INTELIGENCIA ARTIFICIAL

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A N

ALMA ELIZABETH CARDENAS JUAREZ

AGUSTIN ROSALES TORRES

ARMANDO MARIA DE URIARTE OCCELLI

DIRECTOR DE TESIS

ING. ALEJANDRO RAMOS LARIOS

MEXICO, D. F. MARZO DE 1989

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Tabla de Contenido

Indice de Figuras

Introducción

Introducción.

La historia de la computación ha sido breve. Aún cuando se han logrado importantes avances en las áreas que la componen, ciertos aspectos han quedado rezagados dentro de este desarrollo, tal es el caso del diseño, el cual a la fecha presenta deficiencias en las partes que lo componen; por ejemplo, en el campo de la interpretación de información no existe un criterio que unifique metodologías, lo que genera diseños deficientes que requieren de grandes inversiones de tiempo y dinero para su mantenimiento.

Para realizar un diseño útil y eficiente en la elaboración de *software*, es necesario recopilar, ordenar y verificar la información. En el proceso que se realiza para ordenar ésta, aparecen los mayores problemas, ya que es aquí donde los datos son sometidos a interpretaciones que generalmente no coinciden de un diseñador a otro.

La Teoría de la Normalización adquiere relevancia en este campo, ya que a través de ella se busca obtener una metodología que genere modelos que minimicen los problemas en cuanto a las anomalías, tanto de búsqueda como de actualización, que se presentan en el momento de la explotación de los datos.

Debido a lo anterior, resulta importante generar paquetes normalizadores que, basados en la Teoría de la Normalización, proporcionen al diseñador una herramienta que mediante estructuras lógicas y funcionales, agilicen la etapa de diseño y le permitan generar un modelo de datos apegado a sus requerimientos.

Es principio de buena administración que los sistemas se diseñen en función de las necesidades y no que las necesidades se ajusten a los sistemas. Para esto es necesario desarrollarlos con base a una plataforma tecnológica que al mismo tiempo sea vanguardista y permita la utilización de herramientas que faciliten el análisis y diseño de los mismos.

El objetivo de este trabajo se fijó con base en los puntos expuestos anteriormente y se pretende proporcionar una herramienta que facilite una de las etapas del diseño de estructuras de información. La herramienta desarrollada es el Paquete Normalizador.

La Teoría de la Normalización involucra el entendimiento de conceptos formales que fundamentan dicha teoría, los cuales llegan a ser abstractos y complicados, lo que impide su aplicación práctica en el diseño.

Aunque la normalización surgió con el modelo relacional y en teoría se aplica directamente a éste, puede ser transportada a cualquier modelo de datos.

La ventaja fundamental que presenta el Paquete Normalizador desarrollado en el presente trabajo, radica en su aplicación inmediata, puesto que no requiere de conocimientos formales de la Teoría de Normalización y únicamente se necesita conocer el significado de la información y la relación entre los datos (la semántica de los datos).

Pensando en propiciar la utilización de este tipo de herramientas y conscientes de la falta de conocimientos al respecto, este Paquete Normalizador presenta una interface amigable con el usuario, ya que se basa en menúes que en forma interactiva propician la obtención rápida de resultados. Aunado a lo anterior, el paquete opera en dos modalidades , la directa que muestra rápidamente los resultados y la tutorial que explica paso a paso todo el proceso de normalización. Además proporciona ayuda de contexto, la cual evita la memorización de pasos de ejecución y permite que el paquete sea utilizado por cualquier persona.

La estructura del presente trabajo consta de dos partes fundamentales, la primera está constituída por los antecedentes teóricos, que ubican a la Teoría de Normalización dentro del proceso de desarrollo del sistema de información y plantea el porqué se utilizan técnicas de Inteligencia Artificial para el desarrollo de este Paquete Normalizador. La segunda parte presenta en forma detallada el desarrollo del producto.

La primera parte consta de seis capítulos, dentro de los cuales el primero describe la importancia de perfeccionar el manejo de la información en la vida cotidiana, los requerimientos necesarios para la implantación de un sistema de información y su justificación, los nuevos conceptos que han surgido de la consideración de la información como recurso, las características de los conceptos de las nuevas técnicas de programación y manejo de datos y el surgimiento de una tendencia que apoya cada vez más a los sistemas administrativos de las bases de datos.

El capítulo dos describe los objetivos que persigue un sistema de base de datos, sus elementos básicos y los diversos modelos de datos que comúnmente se manejan.

El capítulo tres describe los pasos del diseño de una base de datos, profundizando en el diseño lógico, en el cual se ubica el proceso de normalización.

En el capítulo cuatro se describen los conceptos que utiliza el modelo relacional así como sus principales características.

En el capítulo cinco se presenta el objetivo de la normalización así como la descripción detallada de las diferentes formas normales y los conceptos asociados a cada una de ellas.

El capítulo seis presenta un panorama general de las diversas técnicas de Inteligencia Artificial, se explican las características propias del problema, se describen las más importantes técnicas de búsqueda, se menciona la necesidad de representar el conocimiento con cierto formalismo, se muestran algunas herramientas con las que se hace posible la implantación de sistemas inteligentes, haciendo énfasis en los lenguajes declarativos y como caso específico PROLOG, lenguaje con el que se desarrolló el Paquete Normalizador.

La segunda parte contiene un sólo capítulo, en el cual se analizan las ventajas que brinda el Paquete Normalizador como parte del análisis y diseño de un modelo de información. Se establece que una forma de normalización automática asegura la cobertura de todos los pasos y algoritmos necesarios. También se menciona la relación que tiene el lenguaje de programación PROLOG y el enfoque relacional, presentando las ventajas con que éste cuenta para su utilización, además de un panorama general de su sintaxis. Se describe el Paquete Normalizador en forma detallada, como fue implementado en el presente trabajo, explicando los algoritmos utilizados. Se concluye con la presentación completa del código fuente del Paquete Normalizador, el manual de usuario y las pruebas de ejecución.

Finalmente se presentan las conclusiones logradas como resultado de la realización de este trabajo.

I
Antecedentes Teóricos

1 El Concepto de Información.

1.1 Importancia de la Información.

El avance científico y tecnológico es la característica principal de este siglo. En éste, se ha alcanzado un gran desarrollo tecnológico, así como un notable y paralelo crecimiento del acervo científico.

Uno de los inventos más importantes de este siglo es la computadora, que se ha convertido en una herramienta útil para el manejo de información en áreas como la científica, la administrativa, la industrial y la educativa.

La computadora permite el registro masivo de datos gracias a la alta tecnología electrónica (*Hardware*) y permite el manejo de información a través del desarrollo de aplicaciones (*Software*) que conjuntamente forman un Sistema de Información. Sin embargo, los sistemas de información no se encuentran aislados, sino que se encuentran bajo el marco de alguna organización y son gobernados precisamente por estrategias y objetivos específicos[N,Y].

Por manejo de información entendemos la generación, recopilación, ordenamiento y procesamiento de datos. Los datos son afirmaciones, hechos o eventos que ordenados y presentados de cierta manera proporcionan información. El manejo de información de los tiempos actuales no tiene precedentes, más aún, crece a un ritmo vertiginoso y podemos afirmar que continuará así en el futuro.

La infraestructura requerida para la implantación de sistemas de información, implica gastos considerables de:

- adquisición e instalación del equipo de cómputo,
- operación,
- desarrollo y mantenimiento de aplicaciones,
- soporte técnico y
- mantenimiento al equipo de cómputo (*Hardware*).

La adquisición y desarrollo de sistemas de información se justifica cuando se obtienen beneficios dirigidos hacia los objetivos de las organizaciones.

Al tener control sobre los datos relativos a alguna organización y conocimiento de su entorno, podemos obtener información confiable y oportuna, es decir, podemos extraer información veraz justo en el momento que se necesita.

Las organizaciones cuentan con dos tipos básicos de recursos, que en la práctica se representan por:

- Recursos Humanos (Personal),
- Infraestructura de Capital (Instalaciones, maquinaria, edificios, etc.) y
- Capital de Operación.

La información adquiere por sí sola mayor importancia en la medida que proporciona beneficios a las organizaciones, es entonces cuando la información toma el papel de recurso (recurso informático) que puede ser vital para el desarrollo de las diversas organizaciones.

Podemos decir entonces que:

Buen Aprovechamiento de Recursos = **Obtención de mayores beneficios**

Los objetivos que pretenden alcanzar las organizaciones utilizando el recurso informático son:

1) Contribuir a que las organizaciones ocupen un mejor lugar en el futuro, es decir, el Recurso Informático colabora en la supervivencia y crecimiento de las organizaciones.

2) Proporcionar información oportuna a todos los niveles.

3) Ayudar a la planeación, diseño e implantación de Sistemas de Información ajustados a las estrategias de la organización y no estrategias ajustadas a los sistemas de información.

Los sistemas de información actuales son producto de más de 30 años de investigación y desarrollo de la computación, tanto en tecnología de equipo (*hardware*), como en el desarrollo de técnicas de programación y manipulación de datos (*software*).

1.2 Evolución de los sistemas de información.

A principios de la década de los años sesentas, el punto más importante fue la introducción por parte de CODASYL (*COnference on DAta SYstems Languages*) del compilador del lenguaje COBOL (*COmmon Business Oriented Language*), acompañado por la evolución de unidades de almacenamiento en cinta y la aparición subsecuente de los dispositivos de almacenamiento directo (figura 1).

Sin embargo, al comenzar la explotación de los sistemas de información y el surgimiento de aplicaciones más complejas, se observó la necesidad de agregar al compilador de COBOL paquetes (programas generalizados) que facilitaran, tanto el ordenamiento y clasificación de datos como la generación de reportes (figura 2).

Conforme se fueron desarrollando sistemas aún más complejos y sofisticados surgieron organizaciones lógicas de alto nivel para datos y las aplicaciones comenzaron a interrelacionarse entre sí para ponerse a disposición de un mayor número de usuarios.

Grupos empresariales, proveedores de equipos, casas de *software* y universidades, siguieron de cerca la evolución de los sistemas de información y lograron identificar los problemas que éstos presentaban, visualizando así las características más deseables, como las siguientes:

Tecnología de Información.
Inicio de la década de los sesenta.

Fig. 1

Tecnología de Información.
Mediados de la década de los sesenta.

Fig. 2

Fuente : Cárdenas, A. "Introducción ..."

- Independencia de los datos.
- Inmunidad de modificación a programas de aplicación.
- Estructuras de almacenamiento.
- Estrategias de acceso.
- Flexibilidad para relacionar diferentes tipos de registros.
- Irredundancia de los datos.
- Rendimiento, eficiencia y seguridad de los sistemas de información.

Como productos comerciales, surgieron los Sistemas Generalizados para Manejo de Archivos: GFMS (*Generalized File Management Systems*), Sistemas Generalizados para la Administración de Bases de Datos: GDBMS (*Generalized Data Base Managment Systems*) y Sistemas de Bases de Datos/Telecomunicaciones: DB/DC (*Data Base/Data Communications*).

En 1971, CODASYL presentó un documento acerca de las Bases de Datos, DBTG (*Data Base Task Group*) en el cual quedaron asentadas las bases para el desarrollo de los sistemas DBMS (*Data Base Management System*) subsecuentes.

En el desarrollo de sistemas manejadores de archivos, se presentaron diversas tendencias en cuanto a la tecnología del manejo de datos, como son:

GFMS (Aparece a fines de los sesentas).

Se conciben como sistemas que integran tanto las facilidades de los generadores de reportes, como la de los paquetes de ordenamiento y clasificación de datos, teniendo como característica principal el poder utilizarlos como una herramienta poderosa en la manejo de archivos.

Funcionalmente, un GFMS (figura 3) es capaz de realizar gran parte de las tareas de un programa COBOL, además de llevar el control del

almacenamiento y recopilación de información. Los objetivos principales de un GFMS son la sustitución de lenguajes convencionales de programación en aspectos concernientes a:

- Almacenamiento y recopilación de información.
- Comunicación directa con los métodos de acceso básico del sistema operativo anfitrión.
- Producción de reportes.

El lenguaje de programación de un GFMS es de alto nivel y contiene operadores capaces de realizar ciertas funciones de manera más sencilla y amigable que en un lenguaje de programación convencional como COBOL.

Generalmente los GFMS se enfocan a aplicaciones que requieren grandes volúmenes de flujo de datos (entradas/salidas). Los GFMS son parte importante de la tecnología para el manejo de datos.

GDBMS (Aparece a principios de los setentas)

Simultáneamente al GFMS aparece otro enfoque diferente, los Sistemas Generalizados de Administración de Bases de Datos (GDBMS). Los primeros esfuerzos en esta área se remontan a los años sesentas y surgieron debido a la necesidad de acceso efectivo a grandes archivos por parte de diversos programas de aplicación que además requerían de la integración e interrelación de la información en una base de datos común.

Se observó que las estructuras convencionales de archivos restringían las aplicaciones por sus características limitantes, siendo las principales:

- Una llave única de acceso.
- Un sólo tipo de registros.
- Acceso secuencial.

GFMS (Generalized File Management Systems).

Fines de la década de los sesentas, principio de la de los setentas.

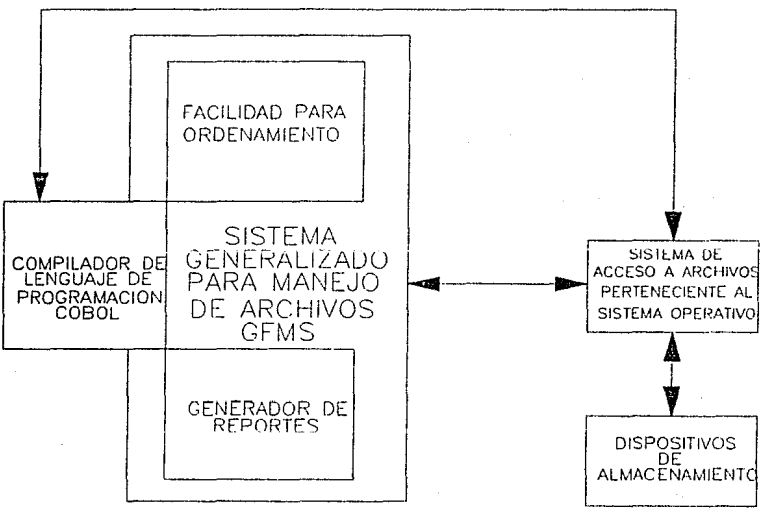


Fig. 3

Fuente : Cárdenas, A. "Introducción ..."

Estas limitaciones eran inadecuadas para aplicaciones más sofisticadas y específicas. Repercutían directamente en tiempo de procesamiento y en excesivos requerimientos de almacenamiento externo.

A su vez, para estas mismas aplicaciones, los GFMS se veían limitados por los métodos de acceso básico de su sistema operativo anfitrión. Es así como surgió el nuevo concepto de organizaciones de alto nivel para archivos GDBMS (figura 4) con las siguientes facilidades:

- Independencia de datos,
- Habilidad de compartir datos,
- Flexibilidad de acceso (lenguaje especializado de consulta),
- Habilidad de relacionar.

DB/DC

Los GDBMS se desarrollaron debido a las limitaciones del sistema operativo anfitrión para el manejo de archivos. Los sistemas generalizados para comunicaciones de datos DC (figura 5) surgieron más recientemente que los GDBMS debido a las limitaciones para el manejo básico de comunicaciones del sistema operativo anfitrión.

Las facilidades proporcionadas por el sistema DC son:

- Independencia de terminales. Permite utilizar cualquier tipo de terminal sin modificar el programa de aplicación.
- Eficiencia y control en la transferencia de grandes volúmenes de datos.
- Control de concurrencia. Acceso simultaneo a la base de datos a través de diferentes terminales y
- Formateo de mensajes y despliegue de información en diferentes terminales.

GDBMS (Generalized Data Base Management Systems).
Desde principios de los años setentas al presente.

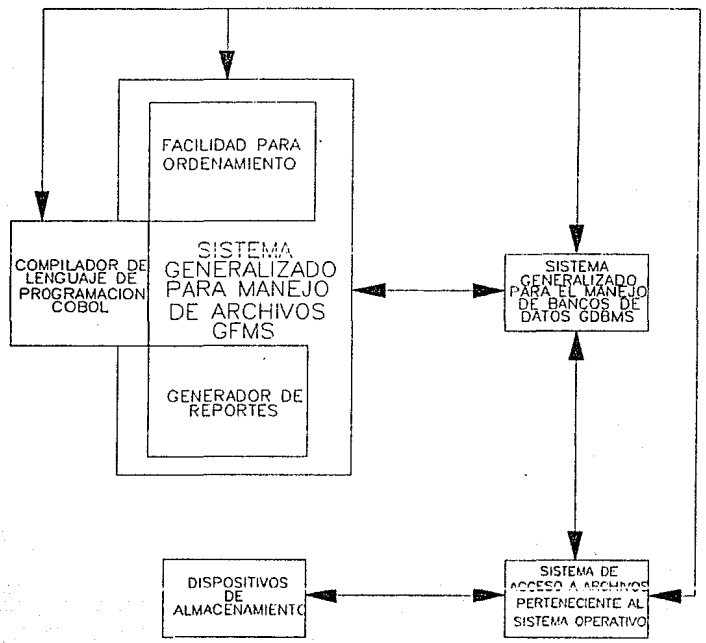


Fig. 4

Fuente : Cárdenas, A. "Introducción ..."

Tecnología de Información sistema DB/DC.

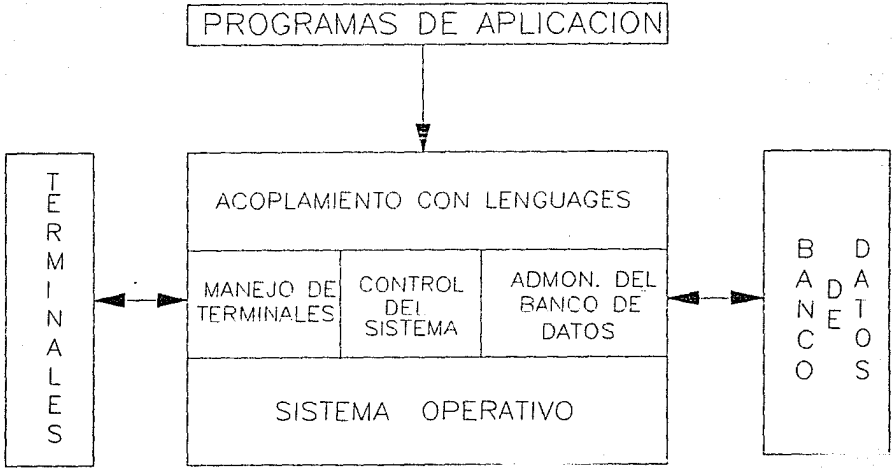


Fig. 5

Fuente : Cárdenas, A. "Introducción ..."

El acoplamiento de un GDBMS (DBMS) y un DC dan como resultado un sistema DB/DC que ofrece las facilidades de ambos.

1.3 Alternativas actuales para el manejo de información.

Nos encontramos entonces con diversas opciones para el manejo de datos, tales como:

- Utilización de lenguajes convencionales de programación (COBOL, PL/1, FORTRAN) para la comunicación con los métodos de acceso del sistema operativo anfitrión, a través de instrucciones estándares del lenguaje.
- Acceso a la base de datos a través del GDBMS desde los programas de aplicación, utilizando lenguajes convencionales de programación.
- Utilización de un lenguaje de consultas particular a cada GDBMS, de muy alto nivel, para lograr accesos rápidos.
- Los programas de aplicación en un lenguaje de GFMS se comunican con los métodos de acceso del sistema operativo anfitrión.
- Los programas de aplicación usando el lenguaje del GFMS para tener acceso a la base de datos mediante el GDBMS.

A pesar del desarrollo de estas nuevas técnicas de manejo de información, aún predomina tanto la utilización de lenguajes de programación convencionales y la utilización de los métodos de acceso básico que proporciona el sistema operativo anfitrión, sin embargo la nueva tendencia se dirige hacia la utilización de los Sistemas Administradores de Bases de Datos (DBMS) y Telecomunicaciones (DB/DC) [F].

2 Bases de Datos.

2.1 Objetivos de un Sistema Manejador de Base de Datos.

La evolución de la organización de información ha originado el uso del término *Base de Datos* como el registro masivo de datos en archivos. Sin embargo, debido a los adelantos en la tecnología de sistemas de información, surge el concepto de Sistema de Base de Datos o en forma general, Sistema Manejador de Base de Datos, siendo una de las alternativas actuales ya mencionada en el capítulo anterior.

Una Base de Datos es una colección de datos integrada, irredundante y que puede compartirse. El conjunto de programas que permite tener una Base de Datos con estas características constituye el Sistema Administrador de la Base de Datos (*Data Base Management System*) [F,Y,N].

Los Sistemas de Base de Datos pretenden lograr, entre otros, los objetivos que a continuación se describen.

2.1.1 Independencia de datos.

La independencia de datos se logra cuando aislamos a los usuarios de los programas de aplicación y a éstos de la Base de Datos, de tal manera que los cambios que surjan en la organización específica de la Base de Datos, tanto a nivel lógico como a nivel físico, no los afecten.

La independencia de datos física aisla los programas de aplicación de los cambios en la organización física de los datos, por ejemplo, cambios en la localización de los dispositivos de almacenamiento o modificación de las rutas de acceso.

La independencia de datos lógica aisla los programas de aplicación de los cambios en la organización lógica de los datos, por ejemplo, adición o eliminación de un campo en la definición de un tipo de registro o modificación de una relación lógica entre registros.

Lo anterior constituye un punto importante, ya que los cambios en sistemas de información son muy frecuentes, debido a requerimientos no previstos y al surgimiento de nuevas necesidades. Cuando los cambios que se realizan no se tienen bajo control, producen serios problemas en los programas de aplicación. Una prueba contundente de este hecho, es el alto costo del mantenimiento del *software*, por lo que cada vez se asigna un presupuesto mayor a la modificación de programas de aplicación [U,P].

La tecnología de Base de Datos pretende lograr tanta independencia de datos como sea posible, sin embargo, en la actualidad no existe un DBMS que proporcione una independencia absoluta de los mismos.

2.1.2 Habilidad para compartir datos e irredundancia de los mismos.

El compartir datos significa que varias aplicaciones tengan acceso a una Base de Datos común.

La redundancia de información consiste en tener una copia total o parcial de los datos que requiere cada aplicación.

La eliminación de redundancia conduce a la irredundancia, que se traduce en la habilidad para compartir datos.

Como consecuencia de la habilidad de compartir datos, el DBMS debe ofrecer la transparencia necesaria para que las aplicaciones puedan operar sin percatarse de la existencia de las demás. Esto ocasiona un mayor tiempo de proceso para el control de concurrencias.

El tiempo de control de concurrencias deteriora el rendimiento (*performance*) del DBMS, llegando a ser crítico en algunas aplicaciones. En estos casos puede ser necesaria cierta redundancia para mejorar el rendimiento.

2.1.3 Habilidad para relacionar datos.

La habilidad para relacionar datos, permite asociar registros o entidades a nivel lógico para obtener información útil (relacionada) y realista sin ambigüedades dentro de un Sistema de Base de Datos.

2.1.4 Integridad.

El término Integridad se refiere a diversas tareas que el DBMS debe realizar para mantener el control y preservar la consistencia de la propia Base de Datos; estas tareas son las siguientes:

- Coordinación del acceso a los datos por diferentes aplicaciones.
- Propagación de los valores actualizados a otras copias o valores dependientes.
- Preservación de un alto grado de consistencia y validez de los datos.

El usuario es responsable de los datos específicos de su aplicación y la consistencia de datos entre aplicaciones es tarea del DBMS.

2.1.5 Flexibilidad de acceso.

La flexibilidad de acceso es la facilidad que el DBMS proporciona para consultar o actualizar los datos de la Base de Datos, mediante cualquier identificador (llave) eliminando la restricción de llave única que imponen los sistemas operativos.

El acceso a los datos puede ser a través de un programa de aplicación, escrito en cualquier lenguaje convencional, aprovechando las facilidades de

acceso del DBMS, o bien, por medio de un lenguaje especializado de consultas similar al lenguaje natural (Inglés, Español, etc.) enfocado a usuarios no programadores.

2.1.6 Seguridad.

Para evitar los accesos accidentales o mal intencionados a la Base de Datos, el DBMS ofrece la facilidad de limitar la vista de la Base de Datos de acuerdo a la función que cada grupo de usuarios desempeña. Este elemento de control es el nivel de seguridad, que determina qué tipo de operaciones (sólo lectura, inserción, borrado, modificación, o alguna combinación de ellas) y sobre qué datos trabajará cada grupo.

2.1.7 Rendimiento y eficiencia.

El gran tamaño de las Bases de Datos y el alto volumen de requerimientos, exigen de un buen funcionamiento de los métodos de almacenamiento, acceso y proceso, que se obtienen de las facilidades del sistema operativo de acuerdo a la relación tiempo / costo.

La viabilidad de un DBMS depende directamente de un rendimiento adecuado y de su eficiencia.

2.2 Elementos básicos de un Sistema Manejador de Bases de Datos.

Una vez identificadas la entidades (tipos de registros) y las relaciones que existen entre ellas (las cuales se analizarán en el Capítulo tres) se requieren de elementos de traducción del concepto lógico de la Base de Datos a la implantación física de la misma.

Estos elementos (figura 6) que son comunes a cualquier DBMS son los siguientes:

DDL (*Data Description Language*):

Es un lenguaje especial de descripción de datos que permite definir la estructura lógica de las entidades y sus relaciones que forman así la Base de Datos. A esta estructura lógica se le conoce como Esquema Lógico de la Base de Datos. A su vez el DDL permite la definición de subconjuntos de la Base de Datos los que se denominan subesquemas, los cuales dan una vista específica de los datos controlando el acceso a la información de cada programa de aplicación.

DML (*Data Manipulation Language*):

Es un lenguaje especial de manejo de datos el cual permite tener acceso a la Base de Datos según los subesquemas definidos.

Las instrucciones del DML se utilizan dentro de los programas de aplicación en lenguajes convencionales o interactivamente a través de lenguajes de consulta.

DMCL (*Device Media Control Language*):

Es un lenguaje que define la estructura física, organizacional y de almacenamiento de la Base de Datos, partiendo de la definición del esquema.

Estructura general de un DBMS.

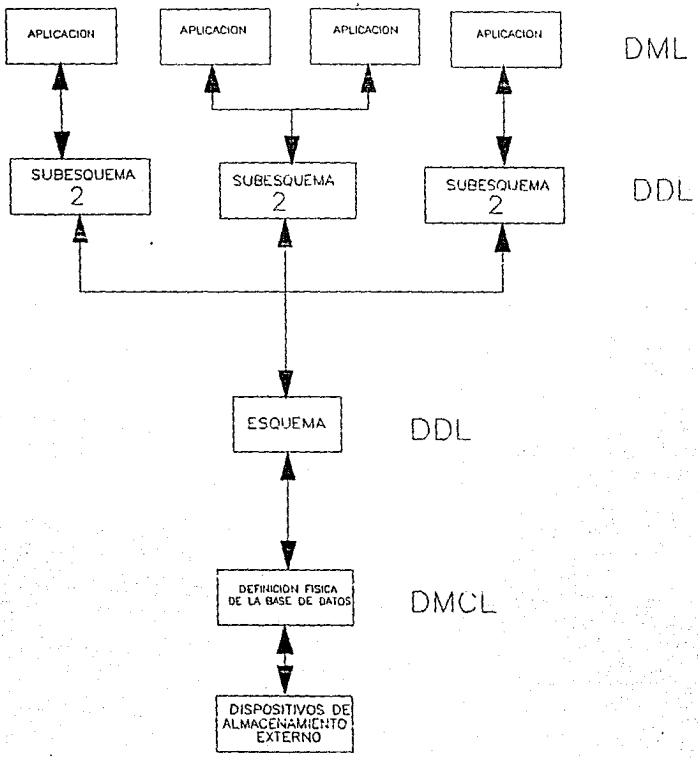


Fig. 6

Fuente : Cárdenas ,A. "Introducción ..."

2.3 Tipos de Bases de Datos.

Existen enfoques alternativos para visualizar y manejar datos a un nivel lógico independientemente de cualquier estructura física de soporte en que se basen.

2.3.1 Modelo Jerárquico

La estructura lógica en la cual se sustenta la Base de Datos jerárquica es el árbol. Un árbol se compone de un nodo raíz y varios nodos sucesores, ordenados jerárquicamente. Cada nodo representa una entidad (tipo de registro) y las relaciones entre entidades son las conexiones entre nodos.

El nodo colocado en la parte superior es llamado padre u *Owner* y los nodos inferiores a estos son llamados hijos o *Members*.

En el sistema jerárquico (figura 7) las conexiones entre archivos no dependen de la información contenida en ellos; las conexiones se definen al inicio y son fijas, entonces, el registro "A" siempre estará ligado al registro "B" no importando cual sea el contenido de éstos y obviamente las conexiones entre tipos de registros también son jerárquicas.

La característica sobresaliente de este modelo es el manejo de la conexión uno-a-muchos, entre un padre y varios hijos, en otras palabras, cada hijo sólo tiene un padre.

En este tipo de Base de Datos la búsqueda de registros padre puede llevarse a través de registros hijos y viceversa, en un buen sistema jerárquico, la actualización de un padre automáticamente actualiza los registros hijos.

Esquema Jerárquico.

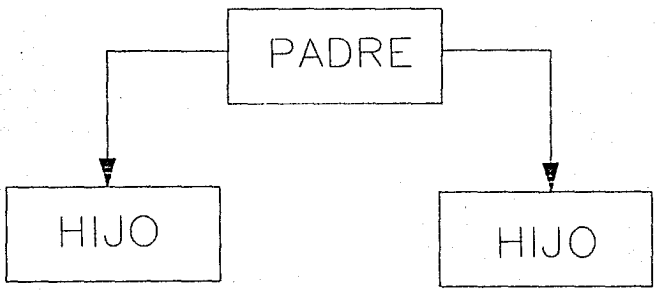


Fig. 7

Fuente : Rich, Elaine "Artificial ..."

2.3.2 Modelo de Redes.

La Base de Datos de Red (figura 8), a diferencia de las jerárquicas, permite cualquier conexión entre entidades, es decir, se pueden representar relaciones de muchos a muchos. En una Red, un hijo puede tener varios padres y varios hijos a su vez.

2.3.3 Modelo Relacional

La estructura lógica de una Base de Datos Relacional está basada en la representación de entidades mediante tablas, las cuales constan de columnas (campos) y renglones (registros). Las relaciones entre tablas se llevan a cabo a través de un conjunto de columnas que se tengan en común, logrando una conexión dinámica entre un número ilimitado de ellas a través del contenido de esas columnas.

La ventaja de los sistemas relacionales es el poder modificar la información sin la preocupación de especificar las combinaciones entre registros.

A continuación se muestra la estructura de una tabla relacional:

RENGLONES UNICOS

COLUMNAS COMUNES

S (S# SNOMBRE STATUS CIUDAD)

S#	SNOMBRE	STATUS	CIUDAD
S1	SMITH	20	LONDRES
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS

P (P# PNOMBRE COLOR PESO CIUDAD)

P#	PNOMBRE	COLOR	PESO	CIUDAD
P1	TUERCA	ROJO	12	LONDRES
P2	TORNILLO	VERDE	17	PARIS
P3	DESARMADOR	AZUL	17	ROMA
P4	DESARMADOR	ROJO	14	LONDRES
P5	LEVA	AZUL	12	PARIS
P6	ENGRANE	ROJO	19	LONDRES

Esquema de Red.

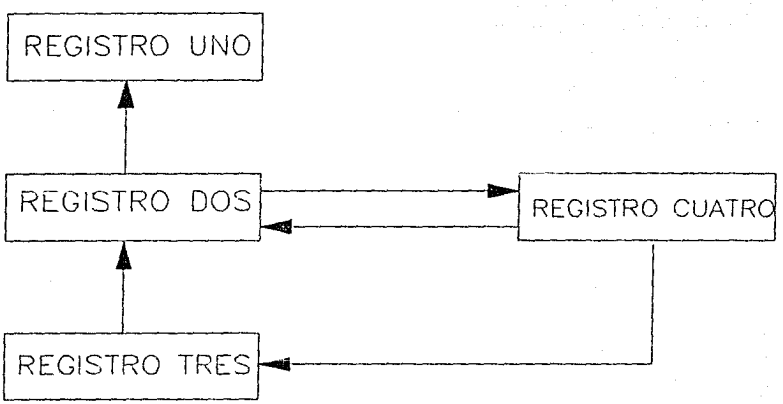


Fig. 8

Fuente: Cárdenas, A. "Introducción ..."

2.3.4 Modelo de Formato Libre.

Este modelo permite alimentar información en cualquier formato: texto, tablas o números y consultarla por medio de llaves de búsqueda. Por ejemplo: se podría redactar un párrafo en referencia a los contactos de algún negocio y definir las llaves de búsqueda **contacto**, **compras** y posteriormente realizar la consulta del párrafo pidiendo información sobre los contactos o compras.

Se puede considerar el modelo de formato libre como relacional, con un formato de registro el cual consiste de dos campos, el párrafo y la llave de búsqueda [S].

3 Proceso de Diseño de una Base de Datos.

En el Capítulo uno se mencionó la importancia que tiene la información para las organizaciones, así como algunas de las alternativas para manejarla.

El diseño de una base de datos se encuentra inmerso dentro del proceso de desarrollo de un sistema. Dicho proceso, pretende la creación de sistemas que produzcan información confiable y oportuna para las organizaciones. Se ha identificado una secuencia general de pasos dentro del proceso de desarrollo [U,P], la cual se enuncia a continuación:

1) Análisis y especificación de requerimientos,

2) Evaluación de alternativas,

3) Diseño e

4) Implantación.

En el primer paso se identifican las necesidades y problemas por resolver, así como las posibles alternativas de solución.

Durante el segundo paso, se analizan las alternativas de solución existentes y en base a ciertos criterios de evaluación, tales como la relación costo/beneficio, se elige la más adecuada. Es aquí donde la base de datos aparece como una posible solución. Asumiendo que se elige una base de datos entonces en el paso de diseño se identifica y define la estructura de la misma, los procedimientos de operación y las especificaciones de programas de aplicación.

En el último paso se efectúa la implantación del sistema, que involucra la instalación de *hardware*, la codificación y pruebas del *software* así como procesos de documentación y entrenamiento a usuarios finales.

Dentro de este proceso el diseño de la base de datos es un paso crítico para alcanzar el éxito en la operación del sistema. El éxito es altamente dependiente de lo preciso de la planeación de la estructura de la base de datos. La mayoría de los problemas de funcionalidad y rendimiento de una base de datos estan asociados a un diseño deficiente [L].

3.1 Etapas del Diseño.

El diseño de una base de datos es un proceso iterativo, para el cual no existe una metodología para su realización, por el contrario, éste es guiado por la experiencia e intuición de los diseñadores [T].

En base a los requerimientos de información, se realiza un diseño preliminar que es revisado conjuntamente con los usuarios, modificándose según sea necesario para obtener el diseño final que permita satisfacer las necesidades.

El diseño debe contemplar tanto las necesidades de los usuarios, como las del equipo de cómputo y las de los programas de aplicación. Es por esto que el diseño de una base de datos se divide en:

- Diseño Lógico y
- Diseño Físico.

El Diseño Lógico toma como punto de partida los requerimientos obtenidos en el análisis, éstos son estudiados para poder identificar y especificar formalmente los registros de la base de datos, su contenido, sus restricciones y sobre todo, identificar la relación entre registros, que es la esencia del concepto de base de datos.

Como resultado se obtiene el esquema lógico de la base de datos que es independiente tanto del equipo de cómputo como del DBMS que se utilice. El esquema lógico refleja de manera formal la estructura de la base de datos.

El Diseño Físico toma el esquema lógico como entrada y lo define físicamente utilizando las herramientas propias de un DBMS específico (como el DDL mencionado en el capítulo anterior), realizando las adecuaciones necesarias para que la base de datos física a través de programas de aplicación o del DML represente fielmente ante los usuarios la estructura lógica.

El diseño físico es totalmente dependiente del DBMS y equipo de cómputo que se utilicen, sin embargo cualquier diseño lógico puede implantarse bajo cualquier DBMS.

3.2 Diseño Lógico.

Existen diversas técnicas para efectuar el Diseño Lógico de una base de datos, que varían desde las que son completamente intuitivas hasta las que involucran procedimientos específicos para el procesamiento del diccionario de datos [T]. Sin embargo en vez de mencionar una técnica específica, creemos de mayor importancia mencionar los puntos críticos que contemplan los elementos necesarios para llegar a obtener un diseño completo.

En el Diseño Lógico se pretenden representar aspectos que son de interés para alguna organización.

Durante el proceso de diseño es necesario tener presente las siguientes consideraciones:

- El nivel de detalle de la representación debe ir de acuerdo con los requerimientos y recursos disponibles.
- El marco dinámico bajo el que se encuentran las organizaciones y que por tanto repercute directamente en su representación.
- La forma en que afectan los eventos que ocurren en el mundo real y como se reflejan en el sistema.

- Los diferentes tipos de usuarios que existen y que conllevan a las diferentes interpretaciones que cada uno de ellos dá a los datos.

Estas consideraciones son importantes ya que permiten mantener la visión global del sistema, puesto que el Diseño Lógico es abstracto y facilmente se puede perder la objetividad.

Como se mencionó anteriormente, el Diseño Lógico obtiene como entrada los resultados del análisis de requerimientos, los cuales se pueden reportar de diversas maneras tales como:

- Diagramas de flujo de información
- Narrativas o
- Diccionario de datos, etc.

El diseñador debe tomar la información contenida en el reporte del análisis y extraer de ahí los conceptos fundamentales para generar el esquema lógico de la base de datos. Para lograr ésto, existen diversos modelos de datos, los cuales tienen diferente orientación. Algunos se orientan en función de la máquina a utilizar y otros de acuerdo a la interpretación humana (figura 9).

Tomando en cuenta que el objetivo principal del Diseño Lógico es representar la realidad y ésta toma significado con la interpretación humana, se ha considerado que el modelo semántico de datos es el más adecuado para identificar y explicar los conceptos del diseño (los cuales son aplicables a cualquier modelo pero con nombres diferentes)[E]. Estos conceptos son los siguientes:

Objetos.

Un objeto es cualquier cosa, lugar o evento que puede ser representado mediante un sustantivo, por ejemplo, una persona es representada por su *nombre*.

Orientación de seis importantes modelos de datos.

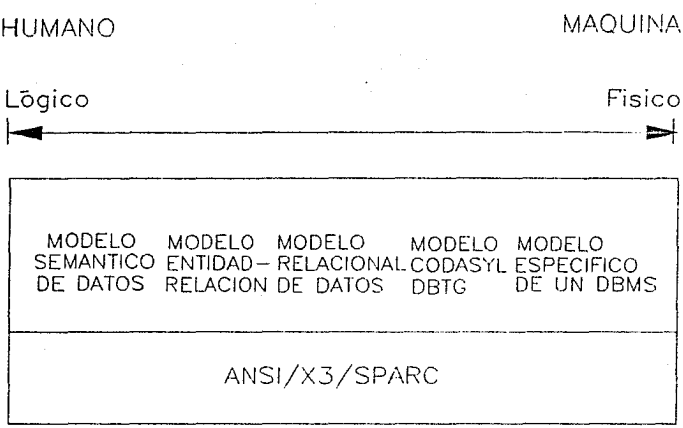


Fig. 9

Fuente : Cárdenas, A. "Introducción ..."

Propiedades de los objetos y sus valores.

Los objetos tienen ciertas características que los diferencían de los demás. Al representar estas propiedades se identifican un conjunto de valores que una propiedad puede tener en un determinado momento, es decir, se identifica el conjunto de valores de las propiedades, por ejemplo, una persona tiene sexo y sus valores son *masculino* o *femenino*.

Clase de objetos.

Los objetos se pueden agrupar al considerar las propiedades que tienen en común y discriminar las diferentes para formar una sola clase de objetos. Al agrupar objetos decimos que generalizamos, por ejemplo, *EMPLEADOS* toda persona que tiene empleo.

Hechos

Es cuando una propiedad de algún objeto toma un valor específico de su conjunto de valores, por ejemplo: Objeto - *persona*; propiedades - *nombre* y *edad*; hecho - *Juan tiene 27 años.*

Asociaciones entre objetos.

Es la relación que existe entre objetos, ya sean de la misma clase o no, por ejemplo: Empleado y Departamento o Empleado y Jefe.

El diseñador analiza los objetos, comúnmente conocidos como entidades, sus propiedades o atributos y sus asociaciones o relaciones. Para

representar formalmente dichos conceptos se pueden utilizar varias técnicas como el modelo Entidad- Relación, el modelo Semántico, etc.. La parte realmente importante consite en que se identifiquen los objetos, sus propiedades y sus relaciones.

Una vez representadas formalmente las entidades y sus relaciones a nivel conceptual podemos iniciar el diseño, siguiendo el siguiente proceso.

1) Identificación de datos por almacenar.

Identificación de datos implica identificar las unidades mínimas de información. El diseñador estudia las propiedades de las entidades para determinar cuales son relevantes para la base de datos. Estas formarán así los datos elementales que contendrá la base de datos.

2) Definición y estandarización de nombres para los datos elementales.

Una vez identificados los datos elementales, es necesario definir los términos que se utilizarán para cada dato elemental.

Como se mencionó, la diferencia entre usuarios provoca una diferencia en la terminología y en su interpretación. Se puede hacer referencia al mismo dato bajo diferentes términos (se identifican sinónimos) o bien, se puede utilizar el mismo término para representar diferentes datos elementales.

Es tarea del diseñador el tratar, en la medida de lo posible, de estandarizar los términos que representarán a los datos elementales para mantener consistencia entre ellos.

La identificación de sinónimos no es una tarea fácil, sobre todo cuando se habla de cientos de datos elementales. Por ejemplo, el dato elemental FECHA, que puede ser FECHA de ingreso de un empleado, FECHA de compra, FECHA de pago, etc.. El diseñador debe definir si se trata del mismo dato elemental o si para efectos del sistema significan eventos diferentes. De ser así, se requerirán de nuevos nombres para referenciar a cada uno de ellos.

3) Desarrollo del Esquema Lógico.

El desarrollo del esquema lógico de una base de datos consiste en definir los registros y sus relaciones. Los registros se definen al determinar que datos elementales contendrán. Al identificar entidades, generalmente se traducen de manera intuitiva en registros. Según Kroenque la identificación de registros es obvia para el 60 ó 75 % de los casos [T]. El diseñador infiere de la especificación de requerimientos algunos registros que deben existir.

Sin embargo, para ciertas entidades es difícil distinguir si deben estar separadas, o bien, deben formar un sólo registro. Esto se debe a que las clases de entidades no son fácilmente identificadas.

La normalización, aunque surgió del concepto relacional de bases de datos[J], es una herramienta de diseño importante para efectos de asignación de datos elementales a registros. [L,E]

De las asociaciones entre entidades que fueron identificadas se definen las relaciones potenciales entre registros. El diseñador debe analizar el ambiente deseado del sistema y las necesidades de los diferentes usuarios para poder discriminar entre las relaciones teóricas y las realmente útiles. Una relación teórica existe lógicamente, pero en la práctica no proporciona información relevante.

Cuando es difícil discernir si la relación puede o no ser útil, es recomendable incluirla dentro del esquema de la base de datos. Siempre se pueden omitir relaciones en el diseño físico, pero agregarlas pude traer serias complicaciones.

La determinación de registros y tipos de relaciones es un ciclo iterativo. Mientras se definen registros, se descubren posibles relaciones y al identificar relaciones se pueden crear nuevos registros. Al analizar conceptualmente entidades, se infieren registros y al encontrar nuevas relaciones se definen nuevas entidades.

Durante este proceso se identifican limitaciones o restricciones que tendrá la base de datos. Estas restricciones generalmente se dividen en tres grupos:

a) Restricciones de datos elementales.

b) Restricciones entre datos elementales dentro de un mismo registro.

c) Restricciones entre datos elementales de diferentes registros.

Estas restricciones se consideran para mantener la integridad de la base de datos. Una restricción del tipo *a)* restringe los valores para cierto dato elemental. Las restricciones del tipo *b)* indican que existe cierta dependencia entre datos elementales o propiedades de una misma entidad o registro. Las restricciones entre elementos de diferentes registros delimitan las relaciones entre estos, obteniéndose así las reglas de integridad de la base de datos.

4) Procesamiento.

La definición del procesamiento establece la forma en que se deberá manejar la base de datos para producir los resultados esperados. El diseñador puede definir el procesamiento describiendo las transacciones y los datos que éstas modifican. Otro método es el realizar diagramas funcionales representando entradas, salidas y funciones específicas de cada programa de aplicación [Z].

Para algunos autores la definición del procesamiento de la base de datos no forma parte del diseño. Sin embargo, una descripción lógica del procesamiento ayuda a identificar flujos propios del diseño lógico.

El diseño concurrente tanto de programas como de la base de datos mejora el diseño de la base de datos y la calidad de los programas [T].

5) Revisión del Diseño.

Una vez obtenido el esquema lógico de la base de datos, es revisado con el propósito de identificar omisiones o malas interpretaciones de los datos. La revisión involucra los requerimientos a los diseñadores y a los usuarios conjuntamente, hasta obtener el diseño que prometa responder a las necesidades planteadas.

A lo largo de este capítulo, se ha enfatizado que el Diseño Lógico es un proceso iterativo, intuitivo y que es difícil de seguir una secuencia rígida de pasos para realizarlo. Esto se debe, en parte, a que al tratar de representar un conjunto de ideas intervienen factores humanos, como el criterio de las personas y su interpretación, los cuales son impredecibles y no se tienen bajo control.

El diseño es una tarea difícil por lo que se han desarrollado una serie de productos para asistirlo como SECSI (*Systéme Expert en Conception de Systémes d'Informations*)[E], *IDMS/Architect*[K].

La normalización de relaciones es una herramienta del diseño por lo que el presente trabajo pretende generar un paquete normalizador que asista a los diseñadores en el proceso de asignación de elementos a las entidades.

Dado que la normalización tiene sus fundamentos en el modelo relacional, consideramos conveniente cubrir los conceptos que giran alrededor del mismo en los siguientes capítulos.

4 Modelo Relacional.

4.1 Conceptos Relacionales.

El Modelo Relacional es el punto intermedio entre los modelos orientados a la interpretación humana y los orientados en función a la máquina a utilizar, ya que tiene características tanto físicas como lógicas.

El Modelo Relacional es lógico debido a que los datos se presentan en un formato familiar a los humanos, pero no contempla la representación física de los datos. Por otro lado, este modelo es más físico que el semántico y que el de entidad relación, ya que el esquema lógico diseñado bajo este modelo, no requiere ser transformado en ningún otro formato dado que existen DBMSs relacionales.

El formato utilizado por el Modelo Relacional es el archivo plano que recibe el nombre de tabla, la cual tiene renglones que representan registros y columnas que representan los campos del registro.

La verdadera importancia del Modelo Relacional no radica en el formato utilizado sino en que las relaciones entre registros se dan a través del valor de los datos.

Para poder hablar del Modelo Relacional debemos conocer algunos términos que nos servirán para diferenciar las partes de dicha estructura:

Atributo.

Se refiere a una columna de una tabla, el cual agrupa datos con el mismo significado semántico.

Tuplo.

Corresponde a un renglón de una tabla, el cual puede contener más de un atributo.

Dominio.

Para poder explicar el concepto de Dominio, hay que partir de la siguiente premisa:

La unidad mínima de información en el modelo relacional, es el valor de un atributo y este valor es atómico, es decir, no se puede dividir[M].

Entonces Dominio es el conjunto de todos los valores posibles de un atributo. Existen dos tipos de dominios: Simples y Compuestos. Los Dominios Simples son aquellos que son indivisibles, por ejemplo, los números enteros. Los Dominios Compuestos son aquellos que se forman de dos o más Dominios Simples, como por ejemplo, las fechas (año, mes, día). Entonces el Dominio Compuesto se forma al realizar el producto cartesiano de los dominios simples (año, mes, día, en ese orden), aunque no todos los valores sean válidos.

Relación.

Una Relación es lo que comúnmente se conoce como tabla y ésta consta de dos partes principales, un encabezado y un cuerpo. El encabezado consta de un conjunto finito de atributos y a cada atributo le corresponde un dominio (aunque no todos los dominios necesariamente son distintos). El cuerpo consta de un conjunto de tuplos, el cual varía con el tiempo. Entonces, cada tuplo está formado por un conjunto finito de atributos apareados con un

elemento del dominio correspondiente. El número de atributos y de tuplos con los que cuenta una relación, nos indican el grado y la cardinalidad de dicha relación. La cardinalidad varía con el tiempo y el grado siempre es fijo. Las Relaciones tienen ciertas propiedades que son consecuencia directa de la definición, éstas son:

1) No existen tuplos duplicados.

El cuerpo de la Relación es un conjunto y en la Teoría de Conjuntos, éstos, por definición no tienen elementos duplicados, por lo que se obtiene el siguiente corolario:

Siempre existe un identificador único (la llave primaria) aunque éste contenga todos los atributos.[M]

2) Los tuplos no están ordenados.

El cuerpo de una relación es un conjunto de tuplos y dado que en la Teoría de Conjuntos el orden no tiene ningún significado, se cumple la propiedad.

3) Los atributos no están ordenados.

Al igual que en el punto anterior el orden de los atributos no tiene ningún significado ya que el encabezado de una Relación es un conjunto de atributos.

4) El valor del atributo es atómico.

Esto se deriva del siguiente hecho: un atributo está relacionado directamente con un dominio y éste tiene valores indivisibles y aunque se piense en dominios compuestos, estos últimos únicamente son concatenaciones de los primeros.

Base de Datos Relacional.

Un usuario puede percibir una Base de Datos Relacional con lo explicado en los puntos anteriores de la siguiente forma:

Colección de relaciones normalizadas de un grado determinado las cuales varían con el tiempo[M].

4.2 Características de una Base de Datos Relacional.

El Modelo Relacional provee un conjunto de operadores algebráicos, para el manejo de datos en todas sus manifestaciones.

Un Sistema Relacional puede dar la idea de una inmejorable forma de organizar la información, pero el Modelo Relacional no es una panacea, dado que un sistema no es necesariamente bueno sólo por soportar este Modelo.

El Modelo Relacional es una herramienta, no un fin en sí mismo, provee fundamentos que pueden ser usados en cualquier sistema, como el lenguaje ensamblador es el fundamento de todo el *software* de hoy en día.

Los sistemas relacionales presentan, entre otras características, una muy importante en la cual la estructura de los datos proporcionados en un modelo, está dada en la propia relación.

Existen diferencias esenciales entre los distintos modelos de base de datos y el relacional. Algunas características importantes del Modelo Relacional, son las siguientes:

- La relación entre tablas, está dada por el valor de los propios datos, eliminando la necesidad del concepto de ligas o apuntadores.
- Simplifica el manejo de información, debido a que ésta se procesa con menos operadores y menor procesamiento colateral.
- Es el primer modelo que introduce conceptos matemáticos para la manipulación y representación de información.

Conociendo todos los aspectos que contempla el Modelo Relacional, podemos decir que un Sistema de Administración de Base de Datos

Relacional (*Relational DBMS*), es aquel que soporta todas las características de éste, aunque algunos aspectos del Modelo Relacional son cruciales y otros son simplemente deseables.

Siguiendo a Codd, definimos un sistema como Relacional sólo si soporta lo siguiente [T]:

- A la Base de Datos Relacional (manejo de Tablas).
- Los operadores *select, project, join* (natural), funcionan sin necesidad de especificar rutas de acceso físico para soportarlos.

Los sistemas que soporten la definición de Codd, aunque no sea explícitamente, pueden calificarse como Sistemas Relacionales.

La justificación de la definición de Codd, puede resumirse en los cuatro puntos siguientes:

1) El Algebra Relacional, tiene una serie de operadores, entre los cuales se encuentran *select, project* y *join*, mismos que forman un subconjunto que permite solucionar casi cualquier problema práctico dentro de las estructuras relacionales.

2) Un sistema que soporta la estructura de datos relacional, pero no a los operadores, elimina la facilidad de uso y productividad de un sistema relacional genuino.

3) Un sistema que soporta los operadores relacionales, pero requiere una predefinición física de ruta de acceso para soportarlos, no provee la independencia física de datos de un sistema relacional verdadero.

4) Para hacer un buen trabajo de implantación de los operadores relacionales, al menos en un ambiente con sistemas de gran capacidad, se requiere que el sistema haga optimaciones. Un sistema que únicamente ejecuta las operaciones requeridas por el usuario, podría ocasionar un bajo rendimiento (*performance*). De esta manera, el implantar un sistema que haga las

operaciones del modelo relacional de manera eficiente, no es una tarea sencilla, por lo que en los productos relacionales es muy importante este punto, el cual, es el reto para mantenerse por largo tiempo en el mercado.

Los aspectos más importantes del Modelo Relacional los podemos englobar en las siguientes divisiones.

4.2.1 Integridad.

El término integridad en el contexto de bases de datos, significa validación y corrección. Con la integridad se evita que los datos dentro de una base de datos sean actualizados inválidamente.

Una actualización inválida se puede deber a lo siguente: errores en los datos de entrada, fallas en el diseño de la estructura de las relaciones, errores del operador o del programador del sistema, fallas del sistema e incluso por falsificación deliberada.

Las reglas de integridad se pueden dividir en dos categorias: Reglas de Integridad de Dominio y Reglas de Integridad Relacional.

Las Reglas de Integridad de Dominio se refieren a la admisión de un valor dado como candidato a un cierto atributo, independientemente de su relación con otro dentro de la base de datos.

Las Reglas de Integridad Relacionales se refieren a la admisión de un tuplo como candidato a una tabla determinando la relación existente de los atributos de ese tuplo con los atributos de alguna otra relación y consta de dos reglas:

- Integridad de Entidades e
- Integridad Referencial.

Estas tienen relación con la llave primaria y las llaves extranjeras, a saber:

Llave Primaria.

La Llave Primaria es un caso especial de lo que se conoce como llaves candidatas. Una Llave Candidata básicamente es un identificador único, pudiendo estar formado por un conjunto de atributos, donde al conocerlos se puede saber el valor de los demás atributos que forman un tuplo, siendo un identificador del tuplo. Por definición, toda relación tiene cuando menos una llave candidata. En una determinada relación se escoge una de las llaves candidatas como Llave Primaria y las demás, si hubieran, pasan a ser llaves alternas.

Sea entonces una relación R con un conjunto de atributos A1..An; estos atributos serán llaves candidatas si y sólo si se cumplen las siguientes propiedades:

a) Unicidad.

En ningún momento dos tuplos de una misma relación podrán tener el mismo valor para el atributo Ai (i = 1..n), es decir, nunca habrá tuplos iguales.

b) Minimalidad.

Ningún atributo Ai (i = 1..n) puede eliminarse sin violar la propiedad de unicidad.

Entonces la Llave Primaria (seleccionanada arbitrariamente de las llaves candidatas) nos permite hacer referencia a los tuplos de dicha relación.

Llave Extranjera.

La Llave Extranjera (que no necesariamente es parte de la llave primaria de la relación que la contiene), es un atributo o un conjunto de ellos en una determinada relación R2, cuyo valor debe de existir en la relación R1, en donde dicho conjunto es llave primaria, aunque R1 y R2 no necesariamente son diferentes.

Entonces la relación entre las llaves extranjeras y las llaves primarias son el factor que mantiene unida la base de datos mediante la relación entre tuplos.

Una vez explicados los conceptos de llave primaria y llave extranjera, podemos plantear las Reglas de Integridad:

a) Integridad de Entidad.

Ningún atributo que forme parte de la llave primaria de alguna relación, podrá aceptar valores nulos.

b) Integridad Referencial.

Si la relación R2 incluye una llave extranjera referenciando la llave primaria de una relación R1, entonces cada valor de la llave extranjera tiene que:

1) Ser igual al valor de la llave primaria en un tuplo de R1 ó

2) Ser totalmente nulo (entiéndase por valor nulo alguno fuera del dominio definido).

En el mundo real, cualquier entidad es distinguible, es decir, tiene una identificación única de cualquier tipo, es por eso que la llave primaria realiza la función de identificador único en el Modelo Relacional, por lo tanto, no puede tener un valor nulo dado que como llave primaria sería una

contradicción, se diría, "Hay una entidad que no tiene identidad", por lo tanto, no existe. De ahí el nombre de integridad de entidad.

El objetivo de la integridad referencial es verificar que el valor de la llave extranjera de alguna relación exista como llave primaria en otra, siempre y cuando la llave extranjera sea no nula.

En un estado dado puede suceder que no se satisfaga alguna de las dos reglas, por lo que ese dato es incorrecto.

Para esto debe existir un subsistema de integridad que se encargue de verificar que las actualizaciones sean válidas y corregir las que no lo sean.

Consideremos la existencia de un subsistema de integridad como un componente de un DBMS con las siguientes responsabilidades:

- Monitorear transacciones u operaciones de actualización y detectar violaciones a la integridad.
- En caso de violación, tomar la acción apropiada como puede ser: rechazar la operación, reportar la violación, corregir el error, etc..

Para que el subsistema de integridad sea capaz de realizar estas funciones, se le tiene que proveer de un conjunto de reglas que definan qué errores revisar, cuándo realizar la verificación y qué hacer en caso de detectar un error.

Un ejemplo de una regla de este tipo sería:

Regla de integridad del proveedor #1 (RIP1)

```
RIP1 :  DESPUES DE ACTUALIZAR S.STATUS:          ---------- 1
          S : STATUS  0                          ---------- 2
        DE LO CONTRARIO
          REGRESA CODIGO "REGLA RIP1 VIOLADA"
          ELIMINA ACTUALIZACION                  ---------- 3
        FIN
```

Esta regla consta de tres partes:

- La parte uno indica el momento de la validación, que dependiendo del tipo de transacción se efectuará antes o después de actualizar la base de datos.
- La parte dos indica la condición a verificar y
- La parte tres indica la acción a tomar en caso de que no se cumpliera la condición.

Las reglas de integridad, expresadas en un lenguaje de alto nivel, son compiladas y guardadas en el diccionario del sistema por el compilador de reglas de integridad.

Se puede definir una nueva regla en cualquier momento y será aceptada mientras el estado actual de la base de datos no viole dicha regla.

4.2.2 Manejo de datos.

4.2.2.1 Algebra Relacional.

El Algebra Relacional es una opción para el manejo de datos en el Modelo Relacional basada en un conjunto de operadores.

Cada operador del Algebra Relacional utiliza una o dos relaciones como entrada y produce otra nueva como salida. Codd originalmente definió ocho operadores que se dividen en dos grupos:[J]

- Las operaciones tradicionales de conjuntos tales como unión, intersección, diferencia y producto cartesiano extendido (considerando por supuesto, que sus operandos son relaciones y no conjuntos).
- Las operaciones relacionales tales como: *select*, *project*, *divide* y *join*.

Cabe hacer notar que el resultado de cada operación algebráica, es una relación, gracias a la propiedad de cerradura que permite generar expresiones anidadas, es decir, hacer el *project* de una unión o el *join* de un *select*, etc..

Todas estas operaciones utilizan dos operandos. Los dos operandos deben ser compatibles, (excepto en el producto cartesiano), es decir, deben ser del mismo grado y el iésimo atributo de cada relación debe estar basado en el mismo dominio. Esta compatibilidad es importante para que se cumpla la propiedad de cerradura y el resultado siga siendo una relación.

A continuación explicaremos más detalladamente las ocho operaciones algebráicas.

Unión.

La Unión de dos relaciones A y B , A UNION B, es el conjunto de tuplos que pertenecen a la relación A, a la relación B o a ambas (figura 10).

Ejemplo:

Sea A el conjunto de proveedores en Londres.

Sea B el conjunto de proveedores de la parte P1.

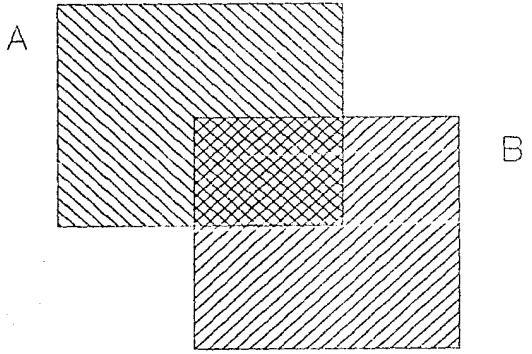
entonces,

A UNION B es el conjunto de proveedores que están en Londres o que proveen la parte P1 o ambas cosas.

Intersección.

La Intersección de dos relaciones A y B, A INTERSECCION B, son todos los tuplos que pertenecen tanto a la relación A como a la relación B (figura 11).

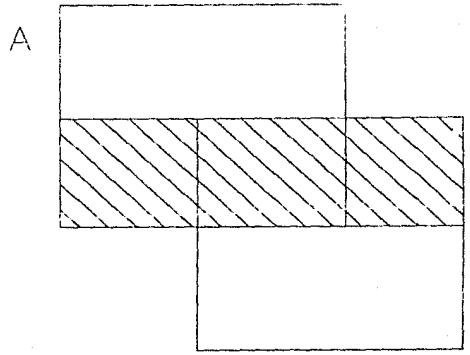
Unión relacional.



A unión B

Fig. 10

Intersección relacional.



A intersección B

Fig. 11

Ejemplo:

Sea A y B igual que el ejemplo de la unión.

entonces,

A INTERSECCION B, es el conjunto de tuplos de proveedores que se localizan en Londres y proveen la parte P1.

Diferencia.

La Diferencia entre dos relaciones A y B, A MENOS B, es el conjunto de tuplos que pertenecen a la relación A y no a la relación B (figura 12).

Ejemplo:

Sea A y B igual que el ejemplo de la unión.

entonces,

A MENOS B es el conjunto de proveedores que se localizan en Londres y no proveen la parte P1.

Producto Cartesiano Extendido.

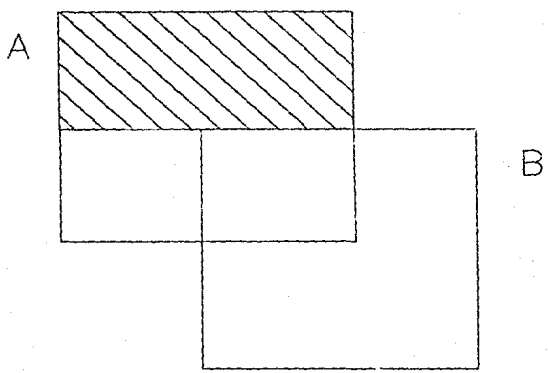
El Producto Cartesiano Extendido de dos relaciones A y B, A POR B, es el conjunto de tuplos t, tal que t es la concatenación del tuplo a, que pertenece a la relación A y un tuplo b, que pertenece a la relación B, para cada tuplo de A, con todos los tuplos de B (figura 13).

Ejemplo:

Sea A el conjunto de números de proveedor.

Sea B el conjunto de números de parte.

Diferencia relacional.



A MENOS B

Fig. 12

Producto cartesiano extendido.

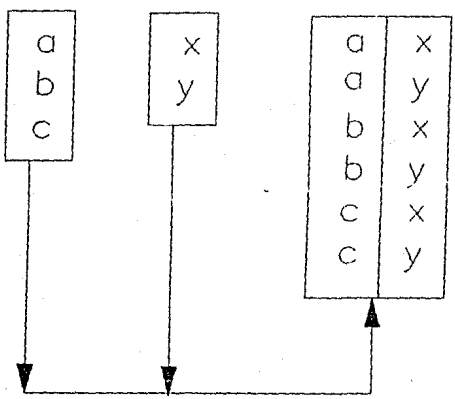


Fig. 13

entonces,

A POR B, es el conjunto de pares, número de proveedor/número de parte, en ese orden y para todo tuplo de A.

Select.

Sea α cualquier operador para comparar valores escalares entonces el *select* (figura 14) sobre la relación R y los atributos X y Y será:

R DONDE R.X α R.Y

Esto dá como resultado el conjunto de tuplos t que pertenecen a R tal que la comparación X α Y evaluada resulta verdadera, es importante notar que tanto X y Y deben ser definidos sobre el mismo dominio y la comparación α debe ser coherente con dicho dominio.

Por lo tanto, la operación de comparación α genera un subconjunto horizontal de dicha relación, esto es, el subjconjunto de tuplos de la relación dada que satisficieron la comparación.

Project.

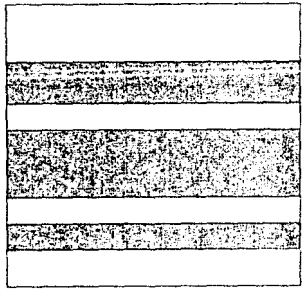
El *project* (figura 15) de la relación R sobre los atributos X, Y...Z

R [X, Y.. Z]

es el conjunto de tuplos (x, y.. z) tal que el tuplo t aparece en la relación R con X-valor-x, Y-valor-y....Z- valor-z, por lo tanto, la operación de *project* genera un subconjunto vertical de la relación dada, esto es, el subconjunto obtenido de la selección de atributos específicos en un orden determinado y de donde se eliminan tuplos redundantes de los atributos seleccionados.

Operación Select.

A

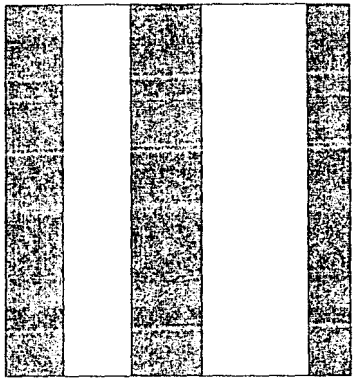


Select

Fig. 14

Operación Project.

B



Project

Fig. 15

Divide.

Esta operación toma como dividendo una relación A de grado (m + n) y como divisor una relación B de grado n y produce una relación como cociente de grado m. Tomando en cuenta que el (m + i)ésimo atributo de A y el iésimo atributo de B (i = 1..n) deben estar definidos en el mismo dominio.

Considerando los primeros m atributos de A como un atributo cualquiera X y el último n como otro Y, tenemos que la relación A se puede considerar como un conjunto de parejas ordenadas de valores (x,y). De la misma forma la relación B puede ser un conjunto de valores (y). Entonces, el resultado de dividir A entre B, (figura 16) esto es, evaluar la expresión:

A DIVIDE B,

es la relación C con un sólo atributo X tal que todo valor x de C.X aparece como valor de A.X y el par de valores (x,y) aparecen en A para todo valor y que aparece en B. Los m atributos del cociente tienen el mismo nombre que los primeros m atributos de la relación B.

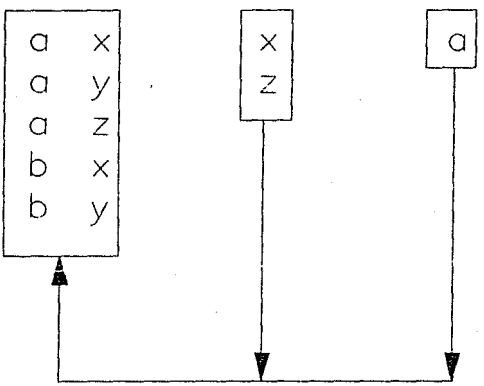
Join.

Sea θ cualquier operador para comparar valores escalares. El Join (figura 17) de la relación A sobre el atributo X con la relación B sobre el atributo Y aplicando θ es el conjunto de todos los tuplos t, tal que t es la concatenación del tuplo a de la relación A y el tuplo b de la relación B y el predicado A.X θ B.Y resulta verdadero (los atributos A.X, B.Y, deben pertenecer al mismo dominio y la operación θ debe ser congruente al dominio).

θ-*Join* no es una operación primitiva, por lo que es equivalente obtener el producto cartesiano extendido con la condición adecuada.

Si el operador θ es la igualdad θ-*Join* recibe el nombre de Equijoin. Partiendo del postulado que dice: El resultado de un equijoin debe tener dos atributos idénticos[M], si uno de esos dos atributos se elimina (mediante un *project*) el resultado recibe el nombre de *Join* Natural; entonces el *Join*

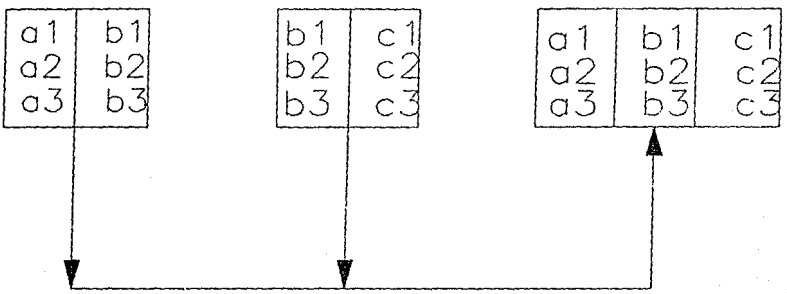
Operación Divide.



DIVIDE

Fig. 16

Operación Join.



Join Natural.

Fig. 17

Natural es el *project* de la restricción de un producto, por lo que el *Join* Natural se convierte en el operador algebráico más importante y más usado.

Entonces A *JOIN* B se define para cualquier atributo común a la relación A y a la relación B, siempre y cuando el dominio de ambos atributos sea el mismo.

4.2.2.2 Cálculo Relacional.

El Cálculo Relacional representa una alternativa para el álgebra relacional como parte manejadora del Modelo Relacional. La diferencia entre el álgebra y el Cálculo Relacional radica en lo siguiente: El álgebra provee un conjunto de operaciones explícitas, tales como el *join*, union, *project*, etc. que pueden ser utilizadas para construir determinadas relaciones de las ya existentes en la base de datos mientras que el Cálculo Relacional únicamente provee una notación para formular la definición de las relaciones deseadas en términos de las relaciones dadas.

En términos generales, cuando menos se puede decir que las formulaciones del Cálculo Relacional son descriptivas y las algebráicas son prescriptivas. El Cálculo simplemente plantea cuál es el problema y el álgebra provee un procedimiento para resolverlo. En otras palabras, el álgebra es procedural y el Cálculo no lo es.

De todas maneras que quede claro que las distinciones son superficiales, por lo que es válido decir que el álgebra y el Cálculo son equivalentes el uno con el otro. Para cada expresión algebráica existe una expresión equivalente en el Cálculo Relacional y viceversa.

Entonces, existe una correspondencia biunívoca entre el Cálculo y el álgebra relacional.

El Cálculo Relacional se fundamenta en una rama de la Lógica Matemática llamada Cálculo de Predicado. La idea de utilizar el cálculo de predicado como base al lenguaje de manejo de datos fue originada por Kuhns y el concepto del Cálculo Relacional fue propuesto por Codd [citados por Date Vol. II]

La notación que definió Codd se muestra a continuación:

Símbolo	Significado
R.D	La relación unaria constituida por el conjunto de valores de datos del dominio D de la relación R.
X(R1.D1,R2.D2..)	Una relación llamada X compuesta de los dominios constituídos por los conjuntos de valores R1.D1,R2.D2...
EXIST	Cuantificador existencial : existe.
FORALL	Cuantificador universal : para todo.
:Tal que.	La expresión a la izquierda de los dos puntos especifica lo que deberá obtenerse de la base de datos y la expresión a la derecha es el predicado o calificativo.
AND,OR,NOT	Los operadores lógicos estándar Y, O, NO para formar calificativos.
=,< >,<,< =,>,> =	Los operadores de comparación estándar: igual a, diferente de, menor que, menor o igual que, mayor que y mayor o igual que.

Una consulta bajo la notación del Cálculo Relacional consta de dos partes: Un objeto (relación destino) que indica los dominios que se desean obtener y una parte calificadora que selecciona los tuplos específicos de la relación destino, dadas las condiciones que se deben satisfacer, por ejemplo:

X(Empleado.nombre,Empleado.edad) :

Empleado.experto_en = "Contabilidad" AND

Empleado.localidad = "Sinaloa"

donde

X(Empleado.nombre,Empleado.edad) es la relación destino que contiene los dominios nombre y edad y Empleado.experto_en = "Contabilidad" AND Empleado.localidad = "Sinaloa" es la parte calificadora que indica la selección.

Un aspecto fundamental del Cálculo es el concepto de Variables de Tuplo *Tuple Variable* también conocidas como variables de rango. Este tipo de variables toman su valor a partir de un rango dentro de una relación, es decir, los únicos valores permitidos son tuplos de la relación, en otras palabras si la variable de tuplo T ejerce su rango sobre la relación R entonces, en cualquier momento T representa algún tuplo t de R.

Por la relación que tiene el Cálculo Relacional con las variables de tuplo éste ha venido a conocerse como Cálculo de Tuplos. Lacroix y Pirrote [citados por Date Vol. II] propusieron una alternativa al Cálculo Relacional la cual llamaron Cálculo de Dominios.

Dentro del Cálculo Relacional las variables de tuplo se definen de la siguiente forma:

RANGE OF T IS X1,X2,.....,Xn

donde T es la variable de tuplo y X1,X2,...,Xn son expresiones que representan relaciones R1,R2,.....,Rn, estas relaciones deben ser compatibles para la unión y los correspondientes atributos deben nombrarse igual en cada relación. La variable de tuplo T actúa sobre la unión de dichas relaciones.

Cada ocurrencia de una variable de tuplo con una WFF (*Well Formed Formula*) puede ser libre o limitada. Por ocurrencia de una variable de tuplo, queremos decir la aparición de la variable dentro de la WFF. Una variable de tuplo T ocurre dentro de una WFF, tanto en el contexto de la referencia del atributo (T.A donde A es un atributo de la relación donde T actúa) como en la variable que sigue después de algún cuantificador.

Analizando la siguiente expresión:

$$EXIST\,x\,(x > 3)$$

donde x actúa sobre el conjunto de los números enteros. La variable limitada x en esta WFF es *dummy*, únicamente sirve para unir el cuantificador con la expresión del paréntesis. La WFF indica que existe algún número entero mayor a tres. El significado de la expresión permanecerá igual aunque se cambie la variable x por y, entonces la siguiente expresión es semánticamente igual a la anterior:

$$EXIST\,y\,(y > 3).$$

Ahora consideremos la siguiente expresión:

$$EXIST\,x\,(x \geq 3)\,AND\,x > 0$$

Aquí existen tres ocurrencias de x referenciando dos variables diferentes las dos primeras ocurrencias son limitadas y podrán ser cambiadas por otra variable sin alterar el significado de la expresión, la tercera ocurrencia es libre y no puede ser cambiada. Entonces de las dos WFF's que se muestran a continuación, la primera es equivalente y la segunda no lo es:

$$\text{EXIST y (y } \geq 3 \text{) AND x } 0$$

$$\text{EXIST y (y } \geq 3 \text{) AND y} \geq 0$$

Entonces una expresión utilizando variables de tuplo es de la siguiente forma:

$$T.A, U.B, \ldots, V.C\ WHERE\ F$$

donde T,U,.....,V son variables de tuplo, A,B,.....,C son los atributos de las relaciones asociadas y F es un WFF conteniendo a T,U,.....,V como variables libres.

El valor de esta expresión es el *project* del subconjunto generado del producto cartesiano extendido T x U x ... x V, cuando F es verdadero.

4.2.2.3 Algebra Relacional vs Cálculo Relacional.

El álgebra relacional y el cálculo relacional son equivalentes. Codd probó que el álgebra es tan poderosa como el cálculo. Esto lo hizo mediante el algoritmo *Codd Reduction Algorithm* por medio del cuál cualquier expresión del cálculo se puede reducir a su equivalente semántico del álgebra relacional (E.F Codd *Relational Completeness of Data Sublanguage*). [citado por Date Vol. II]

Para probar que el cálculo relacional y el álgebra relacional son semánticamente equivalentes, presentamos a continuación un ejemplo del funcionamiento de dicho algoritmo.

Como base de datos para este ejemplo utilizaremos las siguientes relaciones:

S

S#	SNOMBRE	STATUS	CIUDAD (proveedor)
S1	SMITH	20	LONDRES
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDRES
S5	ADAMS	30	ATENAS

P

P#	PNOMBRE	COLOR	PESO	CIUDAD (partes)
P1	TUERCA	ROJO	12	LONDRES
P2	TORNILLO	VERDE	17	PARIS
P3	DESARMADOR	AZUL	17	ROMA
P4	DESARMADOR	ROJO	14	LONDRES
P5	LEVA	AZUL	12	PARIS
P6	ENGRANE	ROJO	19	LONDRES

J

J#	NOMBRE	CIUDAD (proyecto)
J1	CLASIFICADORA	PARIS
J2	PERFORADORA	ROMA
J3	LECTORA	ATENAS
J4	CONSOLA	ATENAS
J5	INTERCALADORA	LONDRES
J6	TERMINAL	OSLO
J7	UNIDAD DE CINTAS	LONDRES

SPJ

S#	P#	J#	CANTIDAD (pedido)
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

Consideremos la siguiente consulta:

DAME NOMBRE Y CIUDAD DE LOS PROVEEDORES QUE PROVEEN CUANDO MENOS UN PROYECTO EN ATENAS CON CUANDO MENOS 50 PIEZAS DE CADA PARTE

La expresión en cálculo relacional para la consulta anterior se expresaría de la siguiente forma:

```
SX.NOMBRE SX.CIUDAD WHERE EXISTS JX FORALL

PX EXISTS SPJX

(JX.CIUDAD    = "ATENAS" AND

    JX.J#     = SPJX.J# AND

   PX.P#      = SPJX.P# AND

   SX.S#      = SPJX.S# AND

      SPJX.CANTIDAD  > 50)
```

donde SX, PX, JX, SPJX son variables de tuplo que actúan o ejercen su rango sobre S, P, J, SPJ, respectivamente; ahora explicaremos paso a paso como la expresión anterior sería evaluada con operadores del álgebra relacional para obtener el resultado esperado.

Paso 1: Para cada variable de tuplo consultar el rango (conjunto de posibles valores para esa variable), en forma restringida. Con la frase en forma restringida, queremos decir que puede haber alguna restricción intrínseca dentro de la cláusula WHERE que pueda utilizarse inmediatamente para eliminar ciertos tuplos; en este caso el conjunto de tuplos se reduce a lo siguiente:

VARIABLE DE TUPLO	RANGO	CANTIDAD
SX	Todos los tuplos de S	5
PX	Todos los tuplos de P	6
JX	Tuplos de J donde ciudad sea igual a "ATENAS"	2
SPJX	Tuplos de SPJ donde cantidad sea mayor a 50	24

Paso 2: Obtenemos el producto cartesiano de los rangos seleccionados del cual obtenemos la siguiente tabla, que se presenta en forma resumida ya que consta de 1440 renglones:

S#	SN	STATU	CIU	P#	PN	COLOR	PESO	CIU	J#	JN	CIU	S#	P#	J#	CANTIDAD
S1	SM	20	LON	P1	TU	ROJO	12	LON	J3	LE	ATE	S1	P1	J1	200
S1	SM	20	LON	P1	TU	ROJO	12	LON	J3	LE	ATE	S1	P1	J4	700
S1	SM	20	LON	P1	TU	ROJO	12	LON	J3	LE	ATE	S1	P3	J1	400
...	.														
...	.														
S1	SM	20	LON	P1	TU	ROJO	12	LON	J4	CO	ATE	S1	P1	J4	700
S2	JO	10	PAR	P3	DE	AZUL	17	ROM	J3	LE	ATE	S2	P3	J3	200
S2	JO	10	PAR	P3	DE	AZUL	17	ROM	J4	CO	ATE	S2	P3	J4	500
S4	CL	20	LON	P6	EN	ROJO	19	LON	J3	LE	ATE	S4	P6	J3	300
S5	AD	30	ATE	P2	TO	VERDE	17	PAR	J4	CO	ATE	S5	P2	J4	100
S5	AD	30	ATE	P1	TU	ROJO	12	LON	J4	CO	ATE	S5	P1	J4	100
S5	AD	30	ATE	P3	DE	AZUL	17	ROM	J4	CO	ATE	S5	P3	J4	200
S5	AD	30	ATE	P4	DE	ROJO	14	LON	J4	CO	ATE	S5	P4	J4	800
S5	AD	30	ATE	P5	LE	AZUL	12	PAR	J4	CO	ATE	S5	P5	J4	400
S5	AD	30	ATE	P6	EN	ROJO	19	LON	J4	CO	ATE	S5	P6	J4	500
S5	AD	30	ATE	P6	EN	ROJO	19	LON	J4	CO	ATE	S5	P5	J7	100
S5	AD	30	ATE	P6	EN	ROJO	19	LON	J4	CO	ATE	S5	P6	J2	200
S5	AD	30	ATE	P6	EN	ROJO	19	LON	J4	CO	ATE	S5	P5	J4	500

Paso 3: Restringimos el producto cartesiano recién obtenido acorde a la cláusula WHERE (equivalente a un join), que en este ejemplo es la siguiente:

JX.J# = SPJX.J# AND

PX.P# = SPJX.P# AND

SX.S# = SPJX.S#

De esta manera eliminamos cualquier tuplo en el cual el valor del proveedor no sea igual al proveedor del pedido y la parte no sea igual a la parte del pedido y el proyecto no sea igual al proyecto del pedido, obteniendo así una tabla con únicamente diez tuplos:

S#	SN	STATU	CIU	P#	PN	COLOR	PESO	CIU	J#	JN	CIU	S#	P#	J#	CANTIDAD
S1	SM	20	LON	P1	TU	ROJO	12	LON	J4	CO	ATE	S1	P1	J4	700
S2	JO	10	PAR	P3	DE	AZUL	17	ROM	J3	LE	ATE	S2	P3	J3	200
S2	JO	10	PAR	P3	DE	AZUL	17	ROM	J4	CO	ATE	S2	P3	J4	500
S4	CL	20	LON	P6	EN	ROJO	19	LON	J3	LE	ATE	S4	P6	J3	300
S5	AD	30	ATE	P2	TO	VERDE	17	PAR	J4	CO	ATE	S5	P2	J4	100
S5	AD	30	ATE	P1	TU	ROJO	12	LON	J4	CO	ATE	S5	P1	J4	100
S5	AD	30	ATE	P3	DE	AZUL	17	ROM	J4	CO	ATE	S5	P3	J4	200
S5	AD	30	ATE	P4	DE	ROJO	14	LON	J4	CO	ATE	S5	P4	J4	800
S5	AD	30	ATE	P5	LE	AZUL	12	PAR	J4	CO	ATE	S5	P5	J4	400
S5	AD	30	ATE	P6	EN	ROJO	19	LON	J4	CO	ATE	S5	P6	J4	500

Paso 4: Aplicamos ahora los cuantificadores de derecha a izquierda, en nuestro ejemplo los cuantificadores son:

EXISTS JX FORALL PX EXISTS SPJX

entonces:

a) Realizamos el project de los atributos SPJ.S#, SPJ.P#, SPJ.J# y SPJ.CANTIDAD, obteniendo la siguiente tabla:

S#	SN	STATU	CIU	P#	PN	COLOR	PESO	CIU	J#	JN	CIU
S1	SM	20	LON	P1	TU	ROJO	12	LON	J4	CO	ATE
S2	JO	10	PAR	P3	DE	AZUL	17	ROM	J3	LE	ATE
S2	JO	10	PAR	P3	DE	AZUL	17	ROM	J4	CO	ATE
S4	CL	20	LON	P6	EN	ROJO	19	LON	J3	LE	ATE
S5	AD	30	ATE	P2	TO	VERDE	17	PAR	J4	CO	ATE
S5	AD	30	ATE	P1	TU	ROJO	12	LON	J4	CO	ATE
S5	AD	30	ATE	P3	DE	AZUL	17	ROM	J4	CO	ATE
S5	AD	30	ATE	P4	DE	ROJO	14	LON	J4	CO	ATE
S5	AD	30	ATE	P5	LE	AZUL	12	PAR	J4	CO	ATE
S5	AD	30	ATE	P6	EN	ROJO	19	LON	J4	CO	ATE

b) Dividimos entre la relación P, y obtenemos:

S#	SNOMBRE	STATUS	CIUDAD	J#	JNOMBRE	CIUDAD
S5	ADAMS	30	ATENAS	J4	CONSOLA	ATENAS

c) Realizamos el *project* de J#, JNOMBRE y CIUDAD generándose la siguiente tabla:

S#	SNOMBRE	STATUS	CIUDAD
S5	ADAMS	30	ATENAS

Paso 5: Realizamos el *project* de la tabla anterior acorde a las especificaciones en la lista de objetivo que en nuestro caso es:

SX.NOMBRE SX.CIUDAD

Obteniendo como resultado la siguiente tabla:

SNOMBRE	CIUDAD
ADAMS	ATENAS

Incidentalmente podemos explicar una de las razones (más no la única) por lo que Codd define precísamente los ocho operadores algebráicos analizados anteriormente: Esos ocho operadores proveen una base conveniente para la posible implantación del cálculo, pero quizá lo más importante es que permiten el análisis de la potencialidad de cualquier lenguaje de base de datos.

Se dice que un lenguaje es completamente relacional si es tan poderoso como el cálculo relacional, esto es, si sus expresiones permiten la definición de cualquier relación por medio de las expresiones del cálculo relacional. Entonces en base al algoritmo de reducción de Codd el álgebra es completamente relacional.

El cálculo de dominio difiere del cálculo de tuplo en que maneja variables de dominio en vez de variables de tuplo las cuales actúan sobre dominios y no sobre relaciones.

Lo importante acerca del cálculo de dominio es que soporta una forma adicional de comparación llamada *Membership* que tiene la forma:

R (term1,term2,.....)

En donde R es una relación y cada término es un par de la forma A:v, donde A es un atributo de R y v es una varible de dominio o una constante. Entonces la expresión es verdadera si y solo si existe un tuplo en R que tenga los valores de los atributos especificados.

5 Normalización.

Con el total de datos a ser representados dentro de cualquier organización, surge el problema y la necesidad del diseño de la base de datos, esto es, decidir la estructura lógica para esos datos, decidir qué relaciones son necesarias y qué atributos deben contener.

En el capítulo anterior se explicaron las características del modelo relacional que se orienta tanto al diseño físico como al diseño lógico de la base de datos.

La Teoría de Normalización forma parte importante del modelo relacional ya que toma en cuenta el comportamiento de los datos[N]. Con algunas relaciones al actualizar datos, se pueden tener consecuencias indeseables a las que se les conoce como anomalías de la base de datos.

El proceso de normalización tiene como objetivo eliminar dichas anomalías al reorganizar las relaciones en cierta forma normal.

El estudio de las anomalías provee una guía útil para el diseño de una base de datos, aunque hay circunstancias en las cuales es razonable no incluir un proceso de normalización completo, pero el único requerimiento obligado es que la relación se encuentre en primera forma normal (1FN), por ejemplo:

Cuando se tienen simulaciones financieras que calculan valores de categorías de ingresos y gastos que pueden resultar de la implantación de estrategias financieras, estas simulaciones contienen muchas dependencias transitivas en forma de columnas de totales, renglones de totales y subtotales, relaciones, proporciones e índices financieros, etc.. En estos casos, no es común que el usuario piense en dividir este modelo en dos partes: una en donde se contengan los cálculos del detalle de los datos y otra donde se contengan totales y las estadísticas financieras, esto es, siguiendo una manera común de programación.

Al estructurar los datos bajo el modelo relacional, así como al eliminar algunos tipos de anomalías, no se debe perder de vista el efecto que se tendrá en lo que respecta a la eficiencia del procesamiento y a los criterios psicológicos y estéticos en la presentación de los datos al usuario.

Por lo anterior es que se considera al modelo relacional como un punto intermedio en la orientación hombre-máquina, ya que facilita la representación lógica de datos así como el análisis de su comportamiento que repercute en su procesamiento.

Sin embargo, la Teoría de la Normalización no soluciona automáticamente los problemas, pero ayuda la familiaridad que se tenga con ésta en el diseño.

5.1 Anomalías.

5.1.1 Anomalías de actualización.

Las anomalías de actualización, son problemas que pueden originarse al añadir, borrar o cambiar tuplos en una relación. Estas anomalías no se originan por la administración relacional de los datos, ya que los tuplos en un modelo relacional pueden no estar almacenados en forma permanente (*store*), sino que se originan por la necesidad de responder a consultas formuladas por el usuario.

Dentro de estas anomalías se incluyen las que se mencionan a continuación:

Anomalías de inserción.

Para dar de alta un tuplo es necesario especificar todos los atributos que lo identifican, por ejemplo:

Una relación R = {Proveedor,Status,Ciudad,Pieza,Cantidad} tiene como identificador: {Proveedor,Pieza} por lo tanto, no se puede dar de alta un tuplo del cual no se conozca el proveedor o la pieza.

Anomalías de borrado.

Esta anomalía se presenta cuando se borra una ocurrencia de una relación, la cual contiene información de un atributo que no queremos borrar, por lo tanto, el tuplo se elimina y adicionalmente perdemos la información de ese atributo. Esto realmente es grave cuando el tuplo borrado contenía la última ocurrencia con información del atributo que no se pretendía borrar.

Anomalías de modificación.

Esta anomalía se presenta cuando existen instancias de atributos repetidos, por ejemplo:

Al tener un departamento con 10 empleados existe un punto crítico al momento de hacer modificaciones a alguna de esas instancias, dado que se tienen varios tuplos con el mismo atributo, pudiendo caer fácilmente en inconsistencias al modificar sólo parte de esos tuplos.

5.1.2 Anomalías de procesamiento.

Existen otras anomalías llamadas anomalías de procesamiento que son detectadas en la operación del modelo implantado, afectando directamente su rendimiento.

Anomalías de entrada.

Se generan cuando el usuario requiere cierta salida de un modelo para la cual deba de proporcionar datos innecesarios a la entrada. La anomalía de entrada es similar a las anomalías que conducen a la 2FN de la administración relacional de los datos.

Anomalías de búsqueda.

Ocurre siempre que tenemos dependencias transitivas (dependencias entre atributos que no forman parte de la llave primaria) en una relación. Esta anomalía se elimina al hacer *projects* para obtener dos relaciones diferentes, por ejemplo:

R = {precio, cantidad, costo}

Si hay algún requerimiento en el que se desea conocer el costo de la producción de acuerdo con una cantidad, es necesario dar diferentes valores de precio hasta hacer coincidir los valores de costo de acuerdo a la cantidad especificada, por lo que es necesario separar la relación anterior en dos diferentes con la siguiente estructura: R1 = {Costo, Cantidad} y R2 = {Cantidad,Precio}.

Anomalías de salida.

Una anomalía de salida es una respuesta no determinada a una consulta de usuario.

Una de las causas de estas anomalías es el tener dos o más atributos idénticos como salida en diferentes modelos o aplicaciones bajo una organización relacional.

5.2 Dependencias Funcionales.

Un concepto necesario para entender las formas normales es el de Dependencia Funcional.

La definición de Dependencia Funcional (DF) es la siguiente:

Dada una relación R, un atributo de R llamado Y es dependiente funcionalmente de un atributo X de R, esto es,

$$R.X. \longrightarrow R.Y.$$

R.X = parte determinante

R.Y = parte determinada.

sí y solo sí, cada valor de X en R es asociado con precisamente un valor Y a la vez. Donde X y Y pueden ser conjuntos de atributos[1].

Algunos atributos son dependientes funcionalmente de otros porque conociendo un valor específico de los primeros se conoce o se tiene acceso al valor específico de cada uno de los atributos dependientes.

Si un atributo X es un candidato a llave de la relación R, que en particular es la llave primaria, entonces, todos los atributos Y de la relación R deben necesariamente ser dependientes funcionales de X (llave primaria).

La Dependencia Funcional proviene de la **semántica** de los datos. El proceso de identificar las dependencias funcionales está formado también al entender qué significan los datos y las relaciones entre ellos. Esto es, el analizar semánticamente diferentes situaciones del mundo real que se pueden presentar en la base de datos.

Para poder comprender la definición de cada una de las formas normales, es necesario aclarar otros conceptos usados para definirlas.

Atributo no llave: Es un atributo que no participa o no forma parte de la llave primaria de una relación específica.

Atributos mutuamente independientes: Dos o más atributos son mutuamente independientes si ninguno de éstos dependen funcionalmente entre sí.

5.3 Formas Normales.

La Teoría de la Normalización, está construída alrededor de un concepto que se conoce como Formas Normales. De hecho, una relación se dice que está en alguna forma normal, sólo si cumple con un cierto conjunto de condiciones para cada caso.

En la Teoría de Normalización se incluyen numerosas formas normales. Originalmente Codd definió la primera forma normal 1FN, la segunda forma normal 2FN y la tercera forma normal 3FN[I].

La definición original que hizo Codd de la 3FN sufrió algunas modificaciones debido a fuertes revisiones hechas por Boyce y el propio Codd [I] dado que algunas relaciones que estaban de acuerdo en la 3FN no lo estaban con la nueva definición. Para evitar confusiones, en la actualidad, la nueva definición es referida como forma normal de Boyce/Codd (FNBC) y la original solamente como 3FN.

Después de ésto Fagin [O] definió una cuarta forma normal 4FN (debido a que en ocasiones a la FNBC se le llamaba 3FN). Más recientemente Fagin otra vez definió otra forma normal que llamó *projection-join normal form* (*PJ/NF*) también conocida como quinta forma normal 5FN [N,T].

En este momento podemos mencionar que la Teoría de la Normalización, es decir, el concepto de formas normales e ideas afines no estan consideradas como parte del modelo relacional como tal, pero constituyen un elemento importante en él.

5.3.1 Primera forma normal (1FN).

Una relación está en 1FN si y sólo si todos los dominios delineados para cada uno de los atributos de la relación contienen únicamente valores atómicos.

Esta definición implica invariablemente una redundancia en los atributos que así lo requieran, dado que es necesario repetir algunos valores de los

atributos para cumplir la condición de valores atómicos sin modificar el contenido original de la información. Estas redundancias en la 1FN pueden ocasionar anomalías de actualización.

Para solucionar ésto, es necesario dividir la relación original en un grupo de relaciones donde la información contenida en cada una de ellas, además de estar relacionada entre sí, se pueda accesar con sólo conocer parte de ella.

Para realizar lo anterior es necesario encontrarse en la segunda forma normal.

5.3.2 Segunda forma normal (2FN).

Una relación R, está en 2FN si y sólo si está en 1FN y todos los atributos no llave son dependientes totalmente de la llave primaria.

El proceso de pasar de 1FN a 2FN, consiste en una serie de *projects* , sobre la tabla en 1FN. Estos *projects* tienen la función de dividir o reducir el número de atributos de la relación, ocasionando con ésto, la generación de una serie de relaciones, las cuáles, cumplen totalmente con la definición de la 2FN.

Hay que tener en cuenta que la serie de *projects* obtenidos son equivalentes a la relación original, en el sentido de que siempre se puede recobrar el estado original, aplicando otra serie de *joins* naturales que contrarresten el efecto de los *projects*.

Esto quiere decir que este proceso es reversible. Este proceso es conocido como descomposición sin pérdida, lo que nos lleva a tener la posibilidad de derivar u obtener cualquier información, tanto de la estructura original como de la nueva estructura, con la diferencia que al tenerla organizada, representa mejor al mundo real, donde ésta tiene sentido.

En ocasiones se puede tener una serie de relaciones en 2FN que presenten características indeseables al momento de hacer actualizaciones o búsquedas de información. Para comprender ésto, es necesario introducir otro concepto. Este concepto es el de Dependencia Funcional Transitiva, la cuál nos dice:

Sea:

R.A. ——→ R.B.

(El atributo B es dependiente funcionalmente de A).

y

R.B. ——→ R.C.

se cumple que

R.A. ——→ R.C.

Este comportamiento entre atributos de una misma relación, ocasiona anomalías de actualización, dado que existen atributos que dependen de otros para poder existir o consultar, sin que éstos sean parte de la llave primaria.

Como sucedió con el cambio de 1FN a 2FN, es necesario aplicar a las relaciones que presenten características de dependencia funcional transitiva, una serie de *projects* para separar atributos y eliminar el efecto mencionado, asegurando con ésto que al actualizar una relación, no se generen anomalías.

Hasta este punto se tienen relaciones en 2FN. Al eliminar dependencias funcionales transitivas, nuestras relaciones automáticamente se encontrarán a salvo de este tipo de anomalías.

5.3.3 Tercera forma normal (3FN).

Una relación R, está en 3FN si y sólo si está en 2FN y todos los atributos no llave, no son dependientes transitivamente de la llave primaria.

Esto es, se está en 3FN cuando además de no existir dependencias transitivas, todos los atributos no llave son totalmente dependientes funcionales de la llave primaria.

En esta forma normal se conserva el hecho de existir un proceso reversible, lo cual nos puede llevar a decir que siempre es posible derivar de una serie de relaciones en 3FN, una serie de relaciones en 2FN.

El nivel de normalización de una relación es cuestión de semántica, de tal modo que no sólo importa el valor de los datos, sino también el significado que tienen. No es posible decir en qué forma normal se encuentra una relación, con sólo ver la tabulación de los datos, sino que es necesario conocer su significado para emitir un juicio correcto. De esta menera el tener las relaciones en 3FN no va a asegurar un manejo adecuado de la base de datos. Para evitar esto, se requiere conocer las dependencias relevantes que existan aún cuando las relaciones se encuentren en 1FN ó 2FN.

5.3.4 Forma normal de Boyce-Codd (FNBC).

La 3FN no es válida en el siguente caso:

Tener múltiples llaves candidatas, donde, esas llaves candidatas son compuestas y tienen al menos un atributo común.

Por lo tanto, la 3FN fue remplazada por una definición más general que se conoce como Forma Normal de Boyce/Codd (FNBC). El caso anterior no sucede comunmente en la realidad, por lo que la tercera forma normal es suficiente para la mayoría de los casos.

La definición de la FNBC es la siguiente:

Una relación R está en FNBC si y sólo si todos los determinantes son una llave candidata.

Es importante notar que se habla no sólo de la llave primaria, sino también de llaves candidatas. Además, que la FNBC es conceptualmente más

sencilla que la 3FN y no se hace referencia explícita a la 1FN ni al concepto de dependencia transitiva. A pesar de ésto, es posible obtener a partir de una relación en estado nativo, una serie de relaciones en FNBC por medio de descomposiciones sin pérdidas, utilizando series de *projects*.

5.3.5 Cuarta forma normal (4FN).

Hasta este momento, se ha mencionado el concepto de dependencia funcional y dependencia transitiva, lo cual es aplicable a relaciones de uno a muchos y muchos a uno. Ahora, cuando tenemos una relación entre atributos de muchos a muchos, surge el problema de la representación de las relaciones, puesto que se llega al límite máximo de una llave primaria en una relación, que es el que esta llave esté formada por todos los atributos de la relación, dado que los atributos entre sí son independientes y están relacionados con un sólo atributo.

Ahora, es necesario introducir un concepto que engloba este tipo de relaciones entre atributos. Este concepto es el de Dependencia Multivaluada (DMV).

La definición de Dependencia Multivaluada es la siguiente:

Dada una relación R con atributos A, B, y C, la dependencia multivaluada (DMV):

R.A.——→ R.B.

(El atributo B depende multivaluadamente de A).

se mantiene o existe, si y sólo si, el conjunto de valores B coinciden con una pareja dada de valores de A y C en R, y ésto depende sólo del valor-de-A y es independiente del valor-de-C. Normalmente donde A, B y C pueden ser compuestos.

Esta definición es aplicable sólo cuando una relación tiene al menos tres atributos. También se hace notar que si la DMV R.A.—→R.B. se mantiene, la DMV R.A.—→R.C. también se mantiene, siguiendo la forma de pares de valores para dos atributos y un conjunto de valores para el tercer atributo esto se representa como:

R.A. ——→ *R.B./R.C.*

Tratando de relacionar la DF y la DMV, podemos decir que una DF es una DMV en la cual, el conjunto de valores dependientes coinciden en un sólo elemento, dado un valor determinado para un par de valores de los otros atributos. De esta manera, una DF siempre es una DMV, pero una DMV nunca será una DF.

Esta relación original la podemos transformar en sus dos *projects*, siguiendo el teorema:

Una relación R con atributos A, B y C, puede ser descompuesta sin pérdidas en sus dos projects R1 (A, B) y R2 (A, C) si y sólo si existen las DMVs R.A. —— R.B./R.C..

Ahora, tenemos ya los elementos necesarios para definir la 4FN:

Una relación R está en 4FN si y sólo si existe una DMV en R: A ——B, donde todos los atributos de R son también dependientes funcionales de A.

En otras palabras, sólo las DFs o DMVs en R son de la forma K —X (es decir, una dependencia funcional de una llave candidata K a muchos otros atributos X). Utilizando otros conceptos para definir la 4FN, podemos definirla como sigue:

R está en 4FN si está en FNBC y todas las DMVs en R son en realidad DFs

5.3.6 Quinta forma normal (5FN).

El concepto de descomposición sin pérdida tiene validez hasta la 4FN dado que esta etapa de normalización se basa en dicho concepto, ésto es, el proceso de reemplazar una relación por dos de sus *projects*.

Existen relaciones que no pueden ser descompuestas sin pérdida con dos de sus *projects*, ya que al manejar la información puede ocasionar resultados indeseables e inexistentes, pero haciendo operaciones sobre los atributos de la relación se pueden llegar a eliminar, aunque pueden ser descompuestas en tres o más de sus *projects*.

Para darle formalidad a este concepto, podemos expresarlo así:

Una relación es n-descomposable para grado (n) > 2,

lo cual indica que la relación en cuestión puede ser descompuesta sin pérdida en n *projects* pero no en m *projects* para m < n.

Lo anterior puede ejemplificarse de la siguiente manera:

Teniendo una relación SPJ entonces:

Si el par (S1, P1) aparece en el *project* SP y

el par (P1, J1) aparece en el *project* PJ y

el par (J1, S1) aparece en el *project* JS

entonces el trío (S1, P1, J1) aparece en SPJ, porque el trío (S1, P1, J1) se obtiene del *join* de SP, PJ, y JS.

Dado que la relación *3-descomposable* SPJ es el resultado de una serie de *joins* de ciertos de sus *projects*, se puede hablar de una Dependencia de *Joins* (DJ) la cual podemos definir de la siguiente manera:

La relación R satisface la dependencia de joins:

join (X, Y, ..., Z)

sí y sólo si R es igual al *Join* de sus *projects* en X, Y, ..., Z, donde X, Y, ...Z son subconjuntos del conjunto de atributos de R.

En ocasiones es necesario evaluar si se debe actuar de acuerdo a la *n-descomposable* dado que al realizar los *projects* necesarios se presentan anomalías.

Existe un teorema el cual nos relaciona la descomposición sin pérdida en una relación, las dependencias multivaluadas y la dependencia de *joins*. Este teorema es el de Fagin el cual contiene el siguiente postulado:

R(A, B, C) satisface la dependencia de joins: join(AB, AC) sí y sólo si ésta satisface el par de DMVs

$$A \twoheadrightarrow B/C$$

Esto es, la relación R con atributos A, B y C puede ser descompuesta sin pérdidas en R1 (A, B) y R2 (A, C) si se mantiene una DMV $A \twoheadrightarrow B/C$ en R.

El teorema puede ser tomado como otra definición de DMV por lo cual podemos decir que una DMV es un caso especial de una dependencia de joins (DJ) o que DJ es una generalización de DMV así como ésta es una generalización de una DF. De ésto se desprende que la DJ es la máxima generalización de dependencia posible dentro del área de relaciones y conceptos de Normalización, hasta el momento, aunque pueden surgir otros tipos de dependencia en el futuro.

Podemos tener el caso en donde una DJ no sea una DMV y por lo tanto tampoco una DF, entonces es recomendable y deseable el descomponer cada relación con el menor número de atributos, ésto es en los *projects* que cumplan estríctamente con las características de una DJ y esta descomposición puede ser repetida hasta llegar a tener una relación en quinta forma normal (5FN).

La 5FN se define de la siguiente manera:

Una relación R está en 5FN si y sólo si cada dependencia de joins en R es una consecuencia de las llaves candidatas de R.

Debido a que la 5FN está basada en el concepto de DJ y éste es un caso general de DMV, una relación en 5FN está automáticamente en 4FN.

Para poder entender la definición de la 5FN es necesario observar las llaves candidatas dentro de cada subconjunto de atributos de cada relación. El comportamiento que siguen es que en cada subconjunto de atributos al menos cuenta con un atributo candidato a llave de la relación original, siendo ésta una característica general para poder llegar a tener una relación en 5FN.

Observando de manera contraria a ésta, dada una relación en 5FN podemos llegar a reconocer u obtener las llaves candidatas observando el comportamiento o distribución de atributos en los *projects* que la forman, aunque puede ser una difícil tarea el reconocer las DJ, dado que la interpretación de DJ, a diferencia de las DMV o DF, no es obvia. Por lo anterior se puede llegar a concluir que se esté en 4FN, pero el llegar a concluir que se esté en 5FN no es aún un método claro, por lo que el intentar llegar a una 5FN es prácticamente un caso en extremo raro y en pocas ocasiones se tiene la necesidad de llegar a realizar estos análisis.

5.4 Proceso de normalización.

A continuación se presentan varios ejemplos para identificar los conceptos antes mencionados y mostrar las anomalías que se presentan en cada caso así como su solución a través de la Normalización.

Como ejemplo de las facilidades del modelo relacional usaremos una base de datos de personal.

Consideremos la siguiente información a manejar:

- El nombre de cada empleado (EMP).
- Lenguajes que el empleado conoce (LNG).
- Años de experiencia en el lenguaje (USO).
- Posición del empleado en la organización (POS).

- Años de experiencia (EXP).
- Sueldo por hora del empleado (SDO).
- Proyecto al que el empleado está asignado (PRY).
- Jefe del empleado (JFE).

Que representada gráficamente se muestra a continuación.

PERSONAL 1.

EMP	(LNG , USO)	POS	EXP	SDO	(PRY , JFE)	
ALMA	COBOL, 3					
	FORTRAN 4	PROGR.	4	25	NOMINA	JOSE
					CONTAB	JUAN
ARMANDO	COBOL, 2					
	PL/1 1					
	RPG 3	PROGR.	3	24	INVENT.	AGUSTIN
AGUSTIN	BASIC 4					
	PASCAL 3	JEFE	2	32	INVENT.	AGUSTIN
LUIS	PASCAL 1	PROGR.	1	25	CONTAB	JUAN
					NOMINA	JOSE

En el encabezado de la estructura, los paréntesis encierran a grupos de datos que pueden repetirse.

Con el diseño de PERSONAL 1, cualquier pregunta de la forma:

¿Dime algo del empleado X?

puede ser respondida fácilmente, pero es mucho más difícil responder a requerimientos como:

- ¿Qué empleados usan el lenguaje L?
- ¿Dime quién es el jefe del proyecto P?
- ¿Qué empleados están asignados al proyecto P?
- Cambiar el jefe del proyecto P por el empleado E.

En nuestro ejemplo, le damos a cada empleado un nombre único, por eso EMP puede usarse como llave en PERSONAL 1. En la práctica, puede haber más de una manera de construír una llave en una relación: Registro Federal de Causantes, número de empleado en la organización, etc..

Para poder utilizar la estructura anterior en el modelo relacional es necesario convertirla en una relación, duplicando los valores no repetitivos EMP, POS, EXP Y SDO para cada combinación de valores y grupos repetitivos (LNG, USO) Y (PRY, JFE), la relación puede ser representada en primera forma normal, como se muestra en PERSONAL 2.

PERSONAL 2.

EMP	LNG	USO	POS	EXP	SDO	PRY	JFE
ALMA	COBOL	3	PROGR.	4	25	NOMINA	JOSE
ALMA	FORTRAN	2	PROGR.	4	25	CONTAB.	JUAN
ARMANDO	COBOL	2	PROGR.	3	24	INVENTARIO	AGUSTIN
ARMANDO	PL/I	1	PROGR.	3	24	INVENTARIO	AGUSTIN
ARMANDO	RPG	3	PROGR.	3	24	INVENTARIO	AGUSTIN
AGUSTIN	BASIC	4	JEFE	2	32	INVENTARIO	AGUSTIN
AGUSTIN	PASCAL	3	JEFE	2	32	INVENTARIO	AGUSTIN
LUIS	PASCAL	1	PROGR.	1	25	CONTAB.	JUAN
LUIS	PASCAL	1	PROGR.	1	25	NOMINA	JOSE

Nótese que en PERSONAL 2, el atributo EMP no es suficiente para identificar un tuplo. Se presentan varios tuplos para los empleados que conocen más de un lenguaje o tienen más de un proyecto asignado.

Una posible solución es considerar la combinación de EMP, LNG Y PRY como llave, ya que estos tres valores juntos son suficientes para identificar un tuplo.

Pudiera parecer de momento que PERSONAL 2 representa un paso hacia atrás, no sólo porque requiere de más espacio que PERSONAL 1, sino porque también se dificulta la respuesta a peticiones como:

- Cambia al empleado E el sueldo S (anomalía de modificación)
- Agrega a la asignación del empleado E el proyecto P. (anomalía de inserción)
- Convierte al empleado E en jefe de proyecto P. (anomalía de modificación)
- Elimina el proyecto Contab sin eliminar uso de Fortran de Alma (anomalía de borrado)

Estos problemas se resuelven con los pasos de normalización que aún quedan por hacerse, tomando en cuenta las siguientes consideraciones semánticas:

- Cada empleado tiene sólo una posición,
- Cada empleado es asignado a un conjunto específico de proyectos,
- Cada empleado tiene un jefe por cada proyecto en el que participa,
- El empleado conoce varios lenguajes.

Para poder eliminar las anomalías que generan las peticiones debemos transformar la relación para que todos los atributos no llave dependan totalmente de ella. En este caso se identifica que los atributos POS, EXP y

SDO, no son dependientes de la llave entera en PERSONAL 2. Estos atributos son determinados sólo por EMP, así que tenemos una nueva relación que llamaremos PERSONAL 3, que contenga sólamente EMP, POS, EXP y SDO, teniendo a EMP como la llave.

Por otro lado JFE se determina por EMP y PRY generándose la relación EMP_PRY que tiene como llave a EMP y PRY.

Así mismo el valor de USO se determina mediante EMP y LNG, por lo que crea una relación llamada ANTECEDENTES, teniendo como llave compuesta a EMP y LNG.

PERSONAL 3.

EMP	POS	EXP	SDO
ALMA	PROGR.	4	25
ARMANDO	PROGR.	3	24
AGUSTIN	JEFE	2	32
LUIS	PROGR.	1	25

ANTECEDENTES

EMP	LNG	USO
ALMA	COBOL	3
ALMA	FORTAN	2
ARMANDO	COBOL	2
ARMANDO	PL/1	1
ARMANDO	RPG	3
AGUSTIN	PASCAL	3
AGUSTIN	BASIC	4
LUIS	PASCAL	1

EMP_PRY

EMP	PRY	JFE
ALMA	NOMINA	JOSE
ALMA	CONTAB	JUAN
ARMANDO	INVENTARIO	AGUSTIN
AGUSTIN	INVENTARIO	AGUSTIN
LUIS	CONTAB	JUAN
LUIS	NOMINA	JOSE

Ahora es posible modificar el sueldo del empleado sin saber sus antecedentes ni el proyecto en el que trabaja, es posible dar de alta un

empleado aunque no tenga asignado un proyecto y al eliminar un proyecto no perderemos información del personal y sus antecedentes.

Separando la relación, se evita tener en el modelo atributos que son dependientes sólo de una parte de la llave. Una relación en primera forma normal que no tiene dependencias parciales sobre la llave, se dice que está en segunda forma normal, cumpliéndose ésto para las tres relaciones.

Todas las relaciones del modelo anterior, están en tercera forma normal, porque no existen dependencias transitivas entre atributos.

Dada la definición de la FNBC donde todos los determinantes son llaves candidatas tenemos que:

- La relación PERSONAL3 tiene como identificador único a EMP, sin determinantes por lo que se cumple la definición.
- La relación ANTECEDENTES, con llave EMP,LNG, tampoco tiene llaves candidatas, estando en FNBC.
- La relación EMP_PRY tiene a sus dos atributos EMP y PRY como llave, el determinante EMP JFE es llave candidata por tanto también se encuentra en FNBC.

Así mismo, la definición de 4FN nos dice que las dependencias multivaluadas deben ser dependencias funcionales por lo que concluimos que las tres relaciones obtenidas estan en 4FN.

La descomposición de PERSONAL en las tres relaciones es una descomposición sin pérdida puesto que a través de varios *joins* se obtiene la relación original.

Los casos en los que se aplica la FNBC y la 4FN se presentan rara vez en la vida real. Sin embargo se seleccionaron los siguientes ejemplos para completar la identificación de los conceptos de normalización.

Desafortunadamente las relaciones en 3FN pueden presentar anomalías, por lo que se hace necesario transformarlas a FNBC, para esto utilizaremos la siguiente relación:

Llave primaria (#EST,MATERIA)

Llave candidata (#EST,MAESTRO)

#EST	MATERIA	MAESTRO
100	MATEMATICAS	CARDENAS
150	PSICOLOGIA	ROSALES
200	MATEMATICAS	URIARTE
250	MATEMATICAS	CARDENAS
300	PSICOLOGIA	RAMOS

Debido a que un maestro está asociado únicamente a una materia pero puede haber varios profesores en cada materia, entonces tenemos la siguiente dependencia funcional:

MAESTRO ——· MATERIA

La relación se encuentra en 1FN por definición ya que no tiene atributos no llave. Tambien está en 2FN y debido a que no tiene dependencias transitivas está en 3FN.

Claramente se puede observar que si eliminamos al estudiante 300, se pierde la información referente a que Ramos es maestro de Psicología (anomalia de borrado). Similarmente, no podemos almacenar el hecho de que Acuña es maestro de Economía hasta que un alumno tome la materia de Economía (anomalía de inserción).

Situaciones como las anteriores nos orillan a transformar la relación a FNBC en donde cada determinante es llave candidata; así es que se generan dos nuevas relaciones que no tendrán anomalías, haciendo los siguientes *projects*:

PROJECT RELACION (#EST,MAESTRO)

PROJECT RELACION (MAESTRO,MATERIA)

MATERIA	MAESTRO
MATEMATICAS	CARDENAS
PSICOLOGIA	ROSALES
MATEMATICAS	URIARTE
PSICOLOGIA	RAMOS

Llave: MAESTRO

#EST	MAESTRO
100	CARDENAS
150	ROSALES
200	URIARTE
250	CARDENAS
300	RAMOS

Llave : #EST

En estas relaciones tenemos que cada determinante es llave candidata, cumpliéndose la FNBC. Esto nos permite dar de baja al estudiante 300 sin perder el hecho de que Ramos es maestro en Psicología, así como nos permite dar de alta a Acuña como maestro en Economía, sin la necesidad esperar por la inscripción de algun estudiante a esa materia.

Independientemente de que una relación se encuentre en FNBC, puede llegar a presentar anomalías si ésta presenta dependencias multivaluadas, como se explicará en el siguiente ejemplo:

Llave primaria (#EST,MATERIA,ACTIVIDAD)

Dependencias multivaluadas:

#EST ⟶ MATERIA

#EST ⟶ ACTIVIDAD

#EST	MATERIA	ACTIVIDAD
100	MUSICA	NATACION
100	CONTABILIDAD	NATACION
100	MUSICA	TENIS
100	CONTABILIDAD	TENIS
150	MATEMATICAS	ATLETISMO

En esta relación todos los atributos son llave por lo que se encuentra en FNBC, sin embargo, presenta deficiencias si el estudiante 100 toma una materia más, pues deberemos de buscar todas sus actividades y agregar un tuplo para cada una de ellas, incluyendo esta nueva materia. En este caso dos tuplos deben de ser agregados. Si no se agregan estos dos tuplos, la semántica de la relación no es congruente, ya que el estudiante pierde una actividad al dar de alta una nueva materia (anomalía de inserción).

Ahora, supóngase que el estudiante 100 deja la actividad tenis, por lo que hay que eliminar 2 tuplos ya que si sólo se elimina uno se mantiene la información de la actividad tenis para ese estudiante.

La solución a este problema es realizar los *projects* que se muestran a continuación:

PROJECT RELACION (#EST,MATERIA)

PROJECT RELACION (#EST,ACTIVIDAD)

#EST	ACTIVIDAD
100	NATACION
100	TENIS
150	ATLETISMO

Llave primaria : (#EST,ACTIVIDAD)

#EST	MATERIA
100	MUSICA
100	CONTABILIDAD
150	MATEMATICAS

Llave primaria : (#EST,MATERIA)

Dado que las relaciones tienen dos atributos, no tienen dependencias multivaluadas, por lo tanto se encuentran en 4FN porque estan en FNBC (todo determinante es parte de la llave) y porque las dependencias son sólo funcionales.

6 Técnicas de Inteligencia Artificial.

6.1 Origen de la Inteligencia Artificial.

Desde el origen de la computación, con el desarrollo de grandes máquinas para efectuar cálculos repetitivos a gran velocidad, surgen una serie de especulaciones concernientes a la posibilidad de crear un nuevo género de máquinas que realizaran actividades "inteligentes".

Aún hoy es difícil dar una definición sobre lo que se entiende por actividad inteligente[O], tampoco existe una definición exacta para el concepto de Inteligencia Artificial.

Sin embargo, podemos decir que la Inteligencia Artificial estudia la forma en que una computadora llegue a realizar tareas que los humanos realizamos mejor[K]. Por el momento no se pretende definir ni lo que es la inteligencia ni lo que se entiende por artificial, si no que, se describe lo que constituye una rama de la computación conocida como Inteligencia Artificial.

Algunos de los primeros problemas estudiados por esta rama de la computación fueron la Teoría de Juegos y la demostración de Teoremas Matemáticos. Samuel en 1963 escribió un programa conocido como *Checkers-Playing* que no sólo jugaba con sus oponentes sino que utilizaba la experiencia adquirida en juegos anteriores para mejorar en juegos posteriores[X]. En el mismo año, Newell escribió un programa llamado *The Logic Theorist* que fue un intento para demostrar Teoremas Matemáticos. Aparentemente, la Teoría de Juegos y la demostración de Teoremas Matemáticos, se realizaban adecuadamente a través de una computadora por su rapidez para analizar todas las posibles rutas de solución y seleccionar la mejor.

Por otro lado, Newell [citado por Rich E., 1983] estudió el tipo de razonamiento que los humanos utilizamos para resolver problemas en general. Como producto de esta investigación, presentó junto con Shaw y Simon el programa GPS (*General Problem Solver*), el cual era aplicable a varias tareas, incluyendo el manejo simbólico de expresiones lógicas.

Hasta este momento, las computadoras ofrecían una adecuada capacidad de memoria y velocidad de respuesta para cierto tipo de problemas, como por

ejemplo, las largas cadenas de operaciones aritméticas en un programa de contabilidad. Sin embargo, el avance que se había alcanzado en materia de investigación, distaba mucho del que se había pronosticado en los inicios de la computación.

Las técnicas hasta ese entonces utilizadas, para la solución de problemas a través de una computadora, presentaban serias limitaciones al aplicarlas a problemas de Inteligencia Artificial. En Teoría de Juegos, por ejemplo, se puede escribir un programa para jugar "Gato", almacenando en memoria un vector con todos los posibles movimientos, requiriendo 3^9 posiciones diferentes, para un juego tan sencillo. (No es difícil imaginar el número de posiciones necesarias para representar todas las posibilidades que presenta un tablero de ajedrez); por lo que fue necesario dar otro enfoque a la solución de problemas ya que no existe computadora alguna que sea capaz de soportar la explosión de combinaciones que presentan algunos problemas.

Los primeros esfuerzos realizados con el estudio de la Teoría de Juegos, la demostración de Teoremas Matemáticos y el estudio del razonamiento humano para resolver problemas, permitieron entender que las actividades inteligentes no son triviales; por el contrario, son muy complejas y difíciles de caracterizar. Esta situación propició el desarrollo de nuevas y diversas técnicas para resolver los diferentes problemas estudiados por la Inteligencia Artificial.

Dadas las características que presentaban algunos problemas, se observó que no era suficiente la representación de información a través de números y carácteres. Uno de los avances más relevantes que se ha obtenido con la Inteligencia Artificial es el nuevo concepto en programación conocido como Procesamiento Simbólico.

El Procesamiento Simbólico es el manejo por computadora de información y conocimiento representados por símbolos. Los símbolos hacen referencia a objetos reales y sus propiedades y se pueden enlazar para representar relaciones entre ellos, como dependencias o jerarquías[V]. Este paso de abstracción en el concepto de información permite representar los objetos involucrados en los problemas de Inteligencia Artificial y manejarlos de alguna manera para llegar a una solución adecuada.

Otra de las importantes conclusiones obtenidas, se refiere a que cualquier actividad inteligente requiere de un buen grado de conocimiento, aunque aparente ser una actividad trivial. El conocimiento guarda las siguientes características que lo hacen difícil de representar:

- Es voluminoso,
- Es difícil de caracterizar,
- Es cambiante.

Gran parte de la investigación de Inteligencia Artificial, se ha dedicado al estudio de métodos que permitan explotar y representar el conocimiento de manera que:

- Permita generalizar, no se pretende representar situaciones individuales, por el contrario, se deben agrupar situaciones que compartan propiedades, para efectos de memoria, tiempos de proceso y acceso.
- Sea intelegible para las personas que proporcionan el conocimiento.
- Sea fácil de modificar.
- Aplicable a tantas situaciones como sea posible.

El estudio de métodos para la representación y manipulación del conocimiento no ha evolucionado aisladamente. Conjuntamente se han desarrollado una amplia gama de técnicas de búsqueda e inferencia adecuadas a las diversas características que presentan los problemas de Inteligencia Artificial.

Comencemos con las técnicas de búsqueda. Podemos representar un problema a través de una gráfica, donde contamos con un nodo inicial y sabemos cual es el nodo meta. Las técnicas de representación del conocimiento se encargan de encontrar la mejor forma de representar cada nodo (punto) y de cómo combinarlos entre sí para que representen un estado completo del problema.

En cuanto a la capacidad de inferir sabemos que al entablar una conversación no es necesario decir todas las ideas y conceptos que van implícitos en alguna expresión, pues asumimos que el oyente entiende estas ideas implícitamente y de manera natural. Sin embargo, ¿cómo podríamos enseñar a una máquina para entender ideas que no se dicen, que por el

contario, se asumen o infieren?. La capacidad de inferir y deducir no se enseña en ningún lado, forma parte de nuestro comportamiento natural y las conclusiones que el ser humano obtiene, las adquiere de su experiencia cotidiana[R].

La propiedad de inferir hechos a partir de cierto conocimiento previo es una característica importante de las actividades inteligentes de los humanos. Los estudios realizados al respecto, por la Inteligencia Artificial, han encontrado que la lógica matemática es una herramienta esencial para el desarrollo de técnicas de inferencia.

Otro aspecto importante en el estudio de esta rama de la computación es la percepción del mundo que nos rodea. El proceso de percepción es difícil, comenzando por el carácter analógico de las señales involucradas en la percepción, ya sean éstas auditivas, visuales o de cualquier otro tipo. La Inteligencia Artificial estudia técnicas de reconocimiento de patrones que permiten a una computadora reconocer, clasificar e interpretar imágenes o sonidos.

Aproximadamente 25 años de investigación en Inteligencia Artificial han permitido entender y desarrollar lo siguiente:

- Técnicas para representar y manejar,
- Métodos de búsqueda adecuados a cada problema específico,
- Herramientas de programación que tengan la capacidad de inferencia y abstracción y
- Tecnología que permita la percepción del mundo que nos rodea.

6.2 Problemas a solucionar mediante la Inteligencia Artificial.

El espectro de los problemas estudiados, así como el de técnicas de solución, es muy amplio y continúa extendiéndose cada vez más (figura 18), sin embargo, todos los diversos problemas, comparten una característica en común: Son difíciles de resolver.

Algunos de los problemas que estudia la Inteligencia Artificial son:

Panorama global de la Inteligencia Artificial.

Fig. 18

Fuente: Torfer, A. "Inteligencia ..." 010, Vol.7 #9 Mayo/87,pag.9

- Resolución de problemas en general.

Es el estudio de metodologías que permitan construir programas para la solución de problemas, basándose en el desarrollo de técnicas de búsqueda y manejo del conocimiento.

- Entendimiento del lenguaje natural.

Desarrollo de técnicas que faciliten la comunicación hombre-máquina-hombre utilizando el Lenguaje Natural.

- Percepción y reconocimiento de patrones.

Hacer computadoras capaces de analizar, extraer, describir e identificar patrones de datos de señales acústicas, visuales o de cualquier otro tipo. Por ejemplo, Análisis/Síntesis de voz.

- Almacenamiento y representación del conocimiento:

Debido a la explosión de información, existe la necesidad de realizar sistemas eficientes que permitan manejar un gran volumen de conocimiento, por lo cual un área importante de la Inteligencia Artificial se dedica a desarrollar técnicas de representación del conocimiento por medio de lenguajes especializados. Los Sistemas Expertos, son ejemplo de la utilización de estas técnicas y hasta ahora, son los únicos Sistemas Inteligentes del mercado.

Areas de desarrollo de los sistemas expertos

- Matemática Simbólica,
- Diagnóstico Médico,
- Análisis Químico,
- Ingeniería de Diseño, etc.

Simulación y Modelado.

Un modelo trata de encontrar una imágen o abstracción de algún fenómeno u objeto, reflejando sus rasgos más importantes y tratando de apegarse al mundo real. Su codificación y utilización para resolver algún problema se denomina simulación. Por ejemplo, simulación del tracto vocal para la síntesis de voz.

Lógica Computacional.

Trata de probar que ciertos hechos son consecuencia de otros. Por ejemplo, un programa analizado desde el punto de vista lógico y no sintáctico.

6.3 Situación actual de la Inteligencia Artificial.

En la actualidad existen dos tendencias dentro de la Inteligencia Artificial, la simulación de la experticia y la simulación del sentido común. En el primer caso se han obtenido resultados alentadores al colocar en el mercado los Sistemas Expertos, sin embargo, en la simulación del sentido común no se han logrado resultados satisfactorios. Es por esto último que para algunos autores la Inteligencia Artificial ha dado todo lo que tiene que dar: Sistemas Expertos [V]. Este hecho ha llamado la atención de fisiólogos, psicólogos y computólogos, para estudiar conjuntamente el comportamiento humano, a fin de poder desarrollar nuevas técnicas que permitan alcanzar los objetivos de la Inteligencia Artificial, así como ampliar y modificar los conceptos que se tienen del ser humano.

Las computadoras de la quinta generación, ambicioso proyecto Japonés, reflejará de forma integral los avances alcanzados en materia de Inteligencia Artificial (figura 19). Este proyecto pretende desarrollar sistemas computacionales rápidos, inteligentes y que tengan la capacidad de:

- Tomar decisiones (comparables a las del ser humano).

- Aprender.
- Soportar una interface a través de voz (Lenguaje Natural)
- Generar automáticamente programas.
- Procesamiento distribuido.

Todo esto, a través de comunidades de Sistemas Expertos, trabajando en paralelo en subproblemas para una tarea específica.

La Inteligencia Artificial ha tenido repercusiones en todas las demás ramas de la computación. Aunque las técnicas de Inteligencia Artificial deben ser diseñadas para mantener las restricciones impuestas por cada problema específico, existe cierto grado de independencia entre los problemas y las técnicas para resolverlos. Es posible aplicar técnicas de Inteligencia Artificial a problemas que no pertenecen al área, como también es posible solucionar problemas de Inteligencia Artificial sin las técnicas desarrolladas para dicho propósito, pero la experiencia ha demostrado que sin la utilización de estas técnicas no se obtienen soluciones eficientes.

El criterio para determinar si se ha tenido éxito en el desarrollo de programas inteligentes, radica principalmente en la comparación de los resultados obtenidos con los objetivos para los cuales fue desarrollado el programa. Este punto se presta a discusiones y ambigüedades puesto que es tan complejo como definir lo que es la Inteligencia.

La Inteligencia Artificial, escasamente tiene 25 años de edad, por lo que creemos que junto con las demás ramas de la computación, aún hay mucho que investigar y desarrollar y se vislumbra un futuro prometedor.

Para algunos autores existe la notable tendencia de implantar sistemas basados en el manejo del conocimiento enfocados a la toma de decisiones, proponiendo una evolución de los sistemas de información(figura 20).

Diagrama conceptual de un sistema de computación de quinta generación.

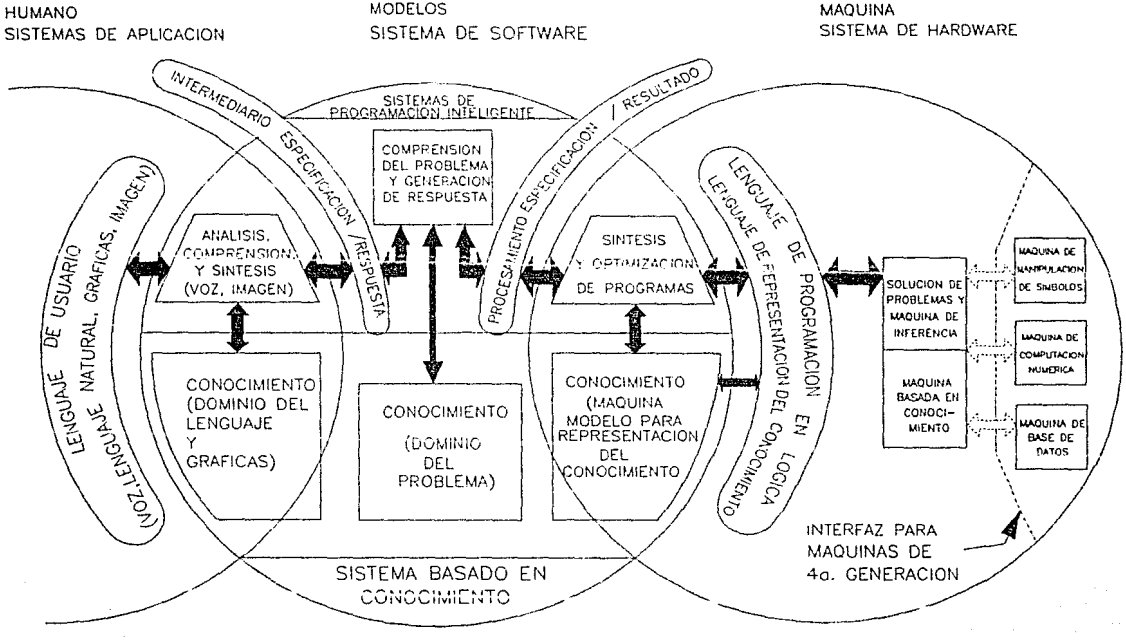


Fig. 19

Fuente: Kowalski, R. "Logic ...", Byte agosto/85

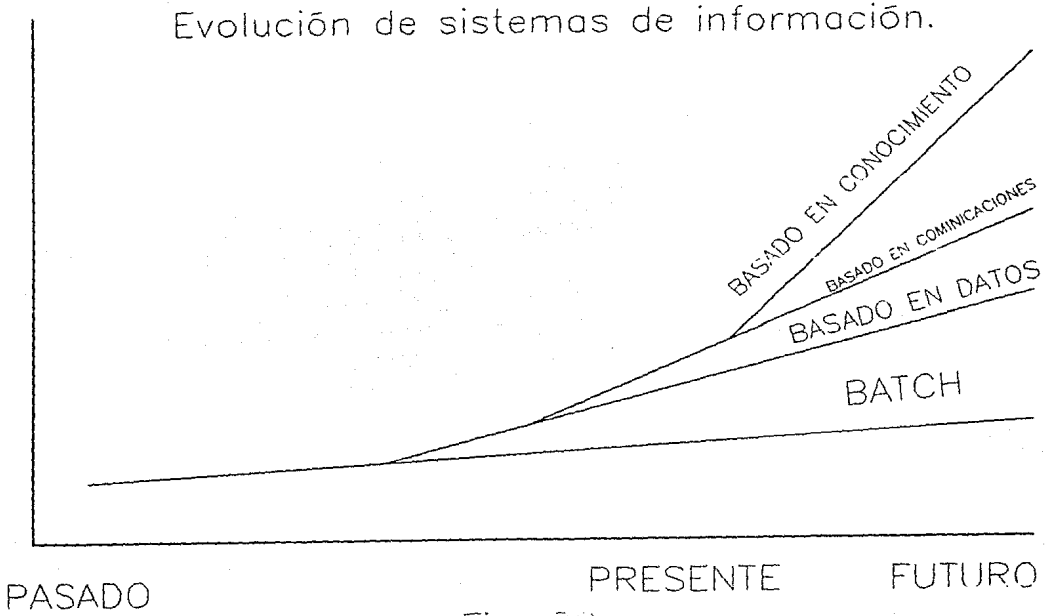


Fig. 20

Fuente: I.B.M. "The Knowledge Based System Center".

6.4 Tipos de problemas y técnicas de solución.

Aunque la gama de problemas estudiados por la Inteligencia Artificial es mucho más amplia que la gama de técnicas de solución, hay unas de carácter general que se explicarán a lo largo de este capítulo.

Existen tres pasos que se deben seguir para resolver cualquier problema mediante la construcción de un sistema:

- Definir el problema con precisión.
- Analizar el problema (obtener características importantes de gran impacto en el momento de escoger una técnica de solución).
- Escoger la mejor técnica y aplicarla.

6.4.1 Definición del problema como un espacio de estados.

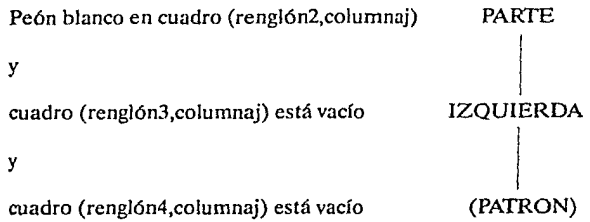
Un problema se puede definir como un espacio de estados. Por ejemplo, en un tablero de ajedrez los estados del problema serían las diferentes posiciones que cada pieza puede tener en combinación con todas las demás.

Aparte de definir los posibles estados del problema, necesitamos encontrar reglas que nos permitan movernos de un estado a otro; para el caso del ajedrez, necesitamos de reglas que definan los movimientos legales que cada una de las piezas puede realizar.

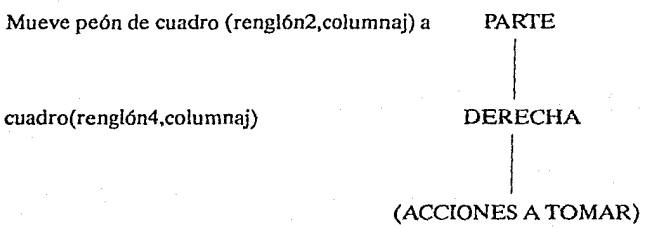
Existe una manera fácil de describir un conjunto de reglas, la cual se forma de dos partes:

- Una parte izquierda; que sirva como patrón a cotejar con el estado actual del problema.
- Una parte derecha; que describa las acciones a realizar para reflejar el movimiento de un estado a otro, por ejemplo:

Para el ajedrez podríamos definir la siguiente regla:



entonces:



Para el ajedrez, aproximadamente existen 10^{120} posiciones diferentes. Si deseamos escribir reglas específicas para cada estado del problema sería un número tan grande de reglas y estados que practicamente imposibilitarían su solución.

El proceso de definición de reglas no es trivial. Se debe buscar la forma de describirlas lo más general posible, pues de otra forma, nos encontraríamos con las siguientes limitaciones:

- Ninguna persona sería capaz de describir un conjunto completo de reglas, debido al tiempo que tomaría y a la sensibilidad a errores.

- Ningún programa podría manejarlas fácilmente, debido a las repercusiones en espacio de almacenamiento, así como en tiempo de proceso de búsqueda.

Mientras las reglas se definan de forma concisa y general, tendremos la capacidad de proveerlas y de elaborar un programa que las maneje.

El espacio de estados de un problema forma las bases para casi todos los métodos de solución de problemas.

La estructura del espacio de estados del problema corresponde a la estructura de la solución del problema en dos formas:

- Permite formalizar la definición del problema, como la necesidad de convertir una situación actual en una situación deseada a través de un conjunto de operadores permisibles.

- Permite definir el proceso de solución de un problema específico, como la combinación de reglas para movernos de un estado a otro y una técnica de búsqueda para explotar el espacio de estados y encontrar una ruta que nos traslade de un estado inicial a otro final o estado solución.

La búsqueda es un proceso importante para encontrar soluciones a problemas difíciles cuando no podemos utilizar técnicas directas.

Además de la definición del espacio y de la descripción de las reglas de movimiento, necesitamos de una estructura de control que permita cotejar entre el estado actual y todas las partes izquierdas de las reglas (patrones), para efectuar la acción descrita en la parte derecha de la regla coincidente y verificar si el estado resultante de la aplicación de la regla es o no la solución.

Resumiendo, podemos obtener una descripción formal del problema en los siguientes pasos:

a) Definir un espacio de estados que contenga todas las posibles configuraciones de los objetos relevantes del problema.

b) Especificar uno o más estados del espacio que describan las posibles situaciones donde el problema puede comenzar. Estos estados se llaman estados iniciales del problema.

c) Especificar uno o más estados que podrían ser soluciones aceptables del problema. Estos estados se conocen como estados meta o estados solución.

d) Especificar un conjunto de reglas que describan las acciones posibles a realizar (operadores). Para este propósito debemos considerar puntos importantes tales como:

- Cuáles son los hechos que asumimos implícitamente en la descripción informal del problema.
- Qué tan generales deben ser las reglas.
- Cuánto del trabajo necesario para resolver el problema, debe ser dedicado para representarlo en reglas.

Una vez definido el problema, éste puede ser resuelto utilizando las reglas en combinación con una estrategia de control, para movernos de los estados iniciales a través del espacio, hasta encontrar una ruta que nos lleve al estado meta o solución.

6.4.2 Análisis del problema.

El proceso de búsqueda es un mecanismo general, fundamental para encontrar soluciones a problemas, sobre todo cuando no existe un método directo de solución.

La búsqueda es el centro de muchos procesos inteligentes, es por esto que los programas de Inteligencia Artificial deben ser estructurados de manera que faciliten la descripción del proceso de búsqueda a través de estrategias de control.

Las estrategias de control deben presentar dos características esenciales:

- provocar movimiento
- ser sistemáticas

como por ejemplo:

a) Construir un árbol con el estado inicial como raíz.

b) Generar todos los nodos sucesores de la raíz aplicando cada una de las reglas aplicables al estado.

c) Para todos los nodos sucesores actuar de acuerdo al punto b).

d) Continuar hasta que alguna regla produzca un estado meta.

A este proceso se le conoce como *Breath-first Search*.

Otra estrategia sistemática de control es la llamada *Depth-first-Search*, la cual expande una rama del árbol hasta que encuentra un estado solución, o bien hasta que alcanza un nivel de expansión predeterminado. Aunque la expansión del árbol puede ser infinita estas dos estrategias permiten el control del movimiento, ambas son sistemáticas y por lo tanto, fácilmente implantables.

Para resolver problemas con una expansión infinita en el árbol, es necesario comprometer los aspectos de movimiento y sistematización para construir una estrategia de control que si bien no garantiza encontrar la mejor solución, al menos garantice obtener alguna. Es así que introducimos el concepto de búsqueda Heurística como una técnica que mejora la eficiencia del proceso de búsqueda.

Hay algunas técnicas heurísticas de propósito general que son aplicables en varios problemas. Como ejemplo, tenemos el algoritmo conocido como *the nearest neighbor* que a partir de un nodo inicial, escoge al mejor nodo sucesor y para cada estado (nodo) escogerá siempre al mejor sucesor. Dependiendo del problema, la función heurística debe evaluar lo que se entiende por mejor y dicha evaluación la dan las características propias del problema.

6.4.3 Características del problema.

Para poder escoger el método más apropiado para un problema específico debemos analizar el problema desde los siguientes puntos de vista:

Analizar si el problema se puede descomponer.

El ejemplo típico de descomposición de problemas es el de integración matemática

$$\int (x^2 + 3x + sen^{2x} \cdot cos^{2x})dx$$

fácilmente podemos descomponer en integrales directas.

$$\int (x^2)dx + \int (3x)dx + \int (sen^{2x} \cdot cos^{2x})dx$$

Sin embargo, hay otro tipo de problemas que no se pueden descomponer. Para los problemas que permiten su descomposición, la técnica divide y vencerás es la que mejor aplica:

- Dividir el problema en subproblemas hasta obtener un subproblema de solución directa o trivial.

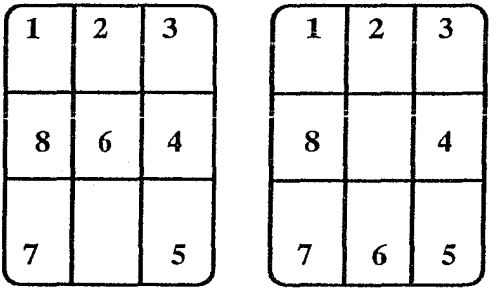
Analizar si las soluciones son reversibles.

Dentro del proceso de búsqueda de solución, al dar un paso, es decir, al movernos de un estado del problema a otro, se puede encontrar que ese paso no lleva a la solución que se pretende encontrar.

Dependiendo del tipo de problema ese paso "mal dado", puede tener repercusiones considerables en la solución. Considérense los siguientes ejemplos:

a) Para probar un teorema matemático, se puede comenzar con algún lema. Eventualmente se descubre que ese lema no ha servido de nada, no obtenemos ninguna repercusión, simplemente se retrocede y se comienza de nuevo con otro lema y el paso realizado puede ser ignorado.

b) Si tenemos el siguiente problema:



ESTADO INICIAL ESTADO FINAL

Se puede cometer un error al dar un paso, por ejemplo, moviendo el número 5 al espacio. Al darse uno cuenta del error, se tiene que retroceder al estado anterior y mover el 6 hacia el espacio.
El paso puede ser recuperable aunque no tan fácilmente como en el ejemplo anterior.

c) En un juego de ajedrez, al dar un paso y darse cuenta hasta dos movimientos después de que ese paso fue erróneo, es un caso en el cual no se puede regresar y modificar el paso realizado. El paso es no recuperable.

Es entonces que se tienen tres tipos de problemas en cuanto a la recuperación de los pasos:

1. Ignorables: Donde los pasos de solución pueden ser ignorados.

2. Recuperables: Donde los pasos de solución permiten regresar y recuperar cierto estado.

3. No recuperables: Donde los pasos de solución no se pueden modificar.

La recuperación de los pasos de solución de un problema, juega un papel importante en la determinación de la estructura de control. Los problemas ignorables pueden utilizar una estructura de control simple. Los problemas recuperables pueden utilizar una estrategia de control un poco más compleja. Los problemas no recuperables necesitan una estrategia que dirige su esfuerzo a tomar decisiones, ya que éstas son definitivas (planeación).

Determinar si el universo del problema es predecible.

En el ejemplo b) del punto anterior, al resolver el problema se sabe exactamente lo que puede pasar. Se tienen todos los elementos para predecir una solución. Otro tipo de problemas, como por ejemplo el Dominó, no son predecibles, ya que cada paso que se tome dependerá de los pasos que den los oponentes; no existe un estado meta bien definido.

Entonces se tienen problemas con dos tipos de universo:

- Predecible
- Impredecible.

En el punto anterior, se menciona que los problemas no recuperables necesitan de un proceso de planeación antes de decidir alguna acción. Sin embargo esto es posible si se trata con un universo predecible.

Los tipos de problemas más difíciles de resolver son los problemas no recuperables de universo impredecible.

Definir si se desea obtener la mejor solución.

Hay algunos problemas donde interesa encontrar una solución sin importar la manera en que se obtiene. Sin embargo, hay otros problemas que por sus características requieren de la manera óptima de llegar al estado meta o mejor solución.

Determinar si la base de datos que almacena el conocimiento es consistente.

Si consideramos que para una demostración matemática la base de conocimientos esta formada por lemas, axiomas y teoremas bien definidos, podemos decir que esta base se encuentra en un mundo consistente.

Si por el contrario, se habla de un analizador semántico de texto, donde existen palabras que dependiendo del contexto tienen diferentes significados, se puede decir que se trata de una base que se encuentra en un mundo de inconsistencias.

Hay esquemas de razonamiento adecuados para los dominios consistentes, como por ejemplo, la lógica matemática, sin embargo éstos serían inapropiados para dominios inconsistentes.

Determinar el papel que juega el conocimiento.

En un juego de ajedrez el conocimiento requerido, como por ejemplo, las reglas para realizar movimientos legales de las piezas, permite dar las restricciones en el proceso de búsqueda de la solución.

Si se trata, por otro lado, de realizar un programa para entender la primera plana de un periódico, se necesitan conocimientos hasta para reconocer lo que sería la solución.

Hay problemas que utilizan conocimiento únicamente, para restringir y guiar el proceso de búsqueda. Sin embargo existen otros tipos de problemas donde el conocimiento es tan importante hasta para poder identificar la solución del problema.

Determinar si se requiere de la interacción de alguna persona en el proceso de solución.

Hay problemas que no requieren interactuar para dar una solución, sin embargo hay otros que necesitan de la interacción del usuario, bien para proporcionar algún elemento de guía en el proceso de búsqueda o para proporcionar información adicional.

Al analizar estos siete puntos clave, dentro del problema a resolver, se puede entonces seleccionar una técnica adecuada para solucionar dicho problema.

6.5 Técnicas de búsqueda.

Antes de explicar algunas técnicas específicas de búsqueda es necesario mencionar cinco aspectos importantes de todas ellas.

1) ¿En qué dirección se debe conducir la búsqueda?.

El objeto del proceso de búsqueda es descubrir una ruta a través del espacio del problema que permita llegar al estado meta a partir de una configuración inicial. La búsqueda puede llevarse a cabo en dos direcciones:

a) Hacia adelante, a partir del estado inicial (figura 21)

Para esto se utiliza el Razonamiento Progresivo cuyos pasos son:

- Generar un árbol cuya raíz es el estado inicial del problema.
- Generar el siguiente nivel del árbol, encontrando todas las reglas cuyos lados izquierdos coincidan con el nodo raíz, utilizando las partes derechas para crear nuevos estados (nodos).
- A cada nuevo nodo creado, se le aplicarán reglas cuyas partes izquierdas coincidan, para generar con las partes derechas, un nuevo nivel del árbol.
- Continuar hasta encontrar un estado que coincida con el estado meta.

b) Hacia atrás, a partir del estado meta (figura 22).

Para esto se utiliza el Razonamiento Regresivo, cuyos pasos son:

- Generar un árbol cuya raíz es el estado meta del problema.
- Generar el siguiente nivel del árbol encontrando todas las reglas cuyas partes derechas coincidan con el nodo raíz (éstas son las únicas reglas que al aplicarlas darían el estado meta que deseamos obtener), utilizar las partes izquierdas de las reglas para generar el siguiente nivel del árbol.
- Para cada nuevo nodo, encontrar las reglas cuyas partes derechas coincidan y generar el siguiente nivel mediante las partes izquierdas de las reglas.
- Continuar hasta que algún nodo coincida con el estado inicial del problema.

Razonamiento progresivo.

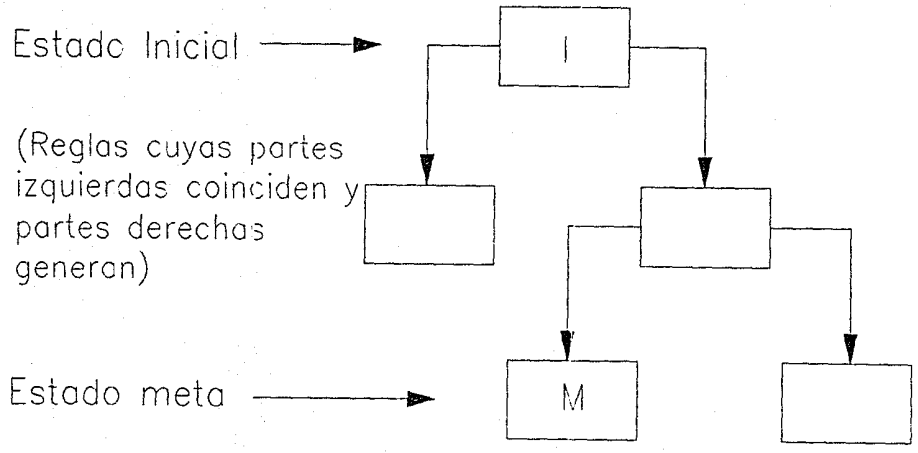


Fig. 21

Fuente : Rich,Elaine "Artificial ..."

Razonamiento regresivo.

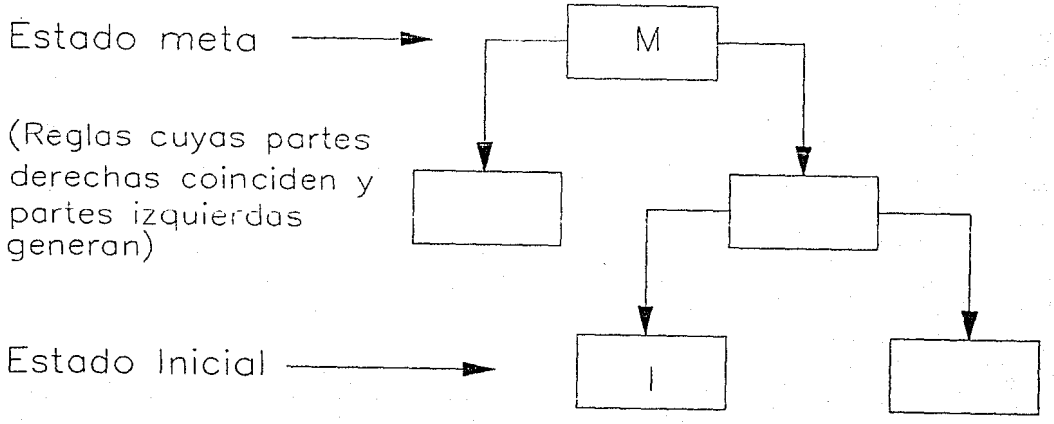


Fig. 22

Fuente : Rich, Elaine "Artificial .."

Cabe hacer notar que el mismo conjunto de reglas permite razonar tanto progresiva como regresivamente.

Para algunos problemas, es conveniente la dirección hacia adelante y para otros, hacia atrás. Podemos mencionar tres factores para determinar la dirección de la búsqueda:

- Número de estados iniciales vs número de estados meta. Es mas fácil iniciar por el menor número de estados.
- Seleccionar la dirección en la que el *Branching Factor* (promedio del número de nodos que genera un solo nodo) es mejor.
- Si es necesario que el programa justifique el proceso de razonamiento al usuario, se debe tomar la dirección que tomaría éste.

Como ejemplo, la demostración de teoremas matamáticos, se inicia con un teorema por demostrar (estado meta) y una buena cantidad de lemas y axiomas que se pueden utilizar. Para este ejemplo el razonamiento regresivo es el adecuado ya que existe un sólo estado meta bien definido y muchos estados iniciales.

Además para este caso el *Branching Factor* es notablemente mayor razonando hacia adelante. A partir de varios axiomas se pueden generar miles de teoremas. Por el contrario, para demostrar un teorema no se necesitan todos los axiomas, sino un número reducido de ellos.

2) Topología del proceso de búsqueda.

Hasta ahora se ha hablado de árboles (figura 23) como la topología que adquieren los problemas en el proceso de búsqueda. Sin embargo, un árbol es un caso particular de una gráfica (figura 24), con la diferencia que en un árbol se pueden tener nodos repetidos, mientras que en una gráfica no, por ejemplo:

Al utilizar una topología de gráfica en el proceso de búsqueda, reducimos el esfuerzo requerido para explorar esencialmente la misma ruta varias veces, pero a su vez, necesitamos esfuerzo adicional cada vez que se genera un nodo al verificar si éste existe o no. Dependiendo del problema este esfuerzo adicional se puede o no justificar. Si el mismo nodo se genera de diferentes maneras es más conveniente utilizar gráficas y si se genera eventualmente, será mas conveniente la utilización de árboles, debido a su simplicidad.

3) Representación de cada nodo en el proceso de búsqueda.

A lo largo de este capítulo se ha descrito el proceso de búsqueda como un mecanismo para viajar alrededor de un árbol o una gráfica donde cada nodo representa un punto en el espacio del problema. Sin embargo, se debe representar a cada nodo individualmente.

En dominios complejos como el mundo que nos rodea, es importante analizar lo siguiente:

- La representación de objetos y hechos individuales.
- La combinación de objetos individuales para obtener una representación completa del espacio del problema.
- La representación eficiente de secuencias de estados del problema que surgen del proceso de búsqueda.

4) Selección de reglas aplicables (*Matching*).

Hasta ahora, no se ha explicado en que forma seleccionar aquellas reglas que pueden ser aplicadas en un cierto estado del problema. Para realizar esta selección se requiere de un tipo de mecanismo que permita cotejar entre el estado actual y las condiciones de las reglas (este punto es crítico en la construcción de sistemas basados en reglas *Rule Based Systems*).

Algunos tipos de *Matching* son los siguientes:

Topología de árbol.

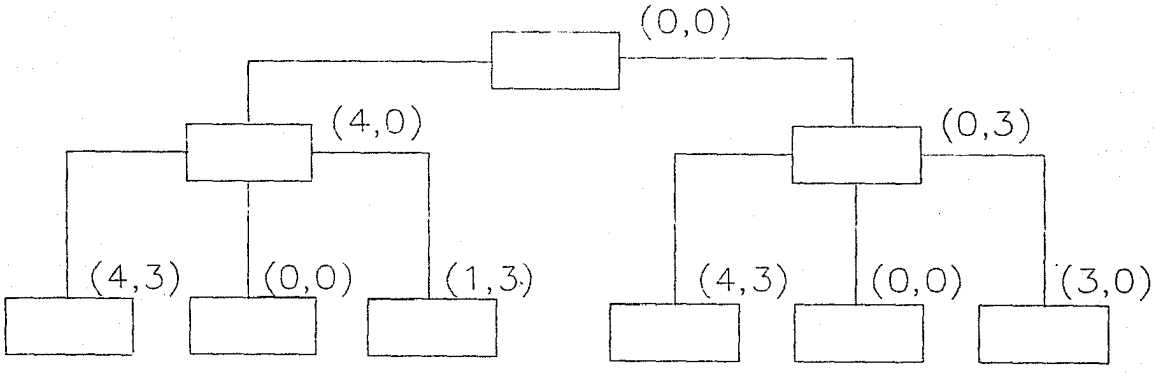


Fig. 23

Fuente : Elaine, R. "Artificial ..."

Topología de gráfica.

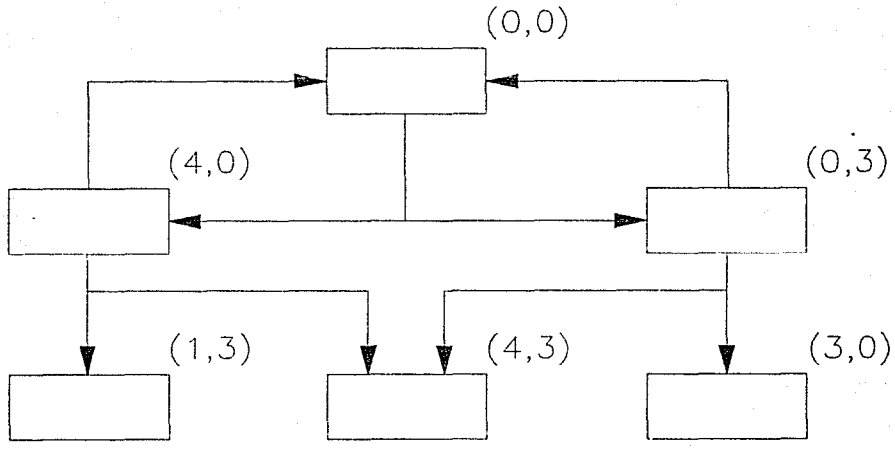


Fig. 24

Fuente : Rich, Elaine " Artificial ..."

- *Matching* indexado.

En vez de buscar en todo el conjunto de reglas, éstas se pueden indexar por medio del estado actual del problema.

- *Matching* con variables.

Cuando se utiliza un *Matching* no literal, no sólo es importante reconocer el *Match* entre el patrón de la regla y el estado del problema, sino también el saber qué asignaciones de variables fueron hechas justo en el momento en el que ocurrió el *Match*.

- *Matching* por aproximación.

Cuando hay reglas que contienen precondiciones no explícitas en algún estado del problema, se requieren de mecanismos de *Matching* que permitan inferir precondiciones. Este *Matching* se vuelve más complejo cuando se tienen aproximaciones a las precondiciones de las reglas, sobre todo cuando se trabaja con descripciones físicas del mundo que nos rodea (la regla más parecida).

El resultado del proceso de *Matching* es una lista de las reglas cuyas partes izquierdas coinciden con el estado actual del problema. El método de búsqueda puede entonces escoger entre todas la regla específica y el orden en que realmente se aplicará.

La manera en la que interactúan el proceso de *Matching* y el de búsqueda depende de la estructura de ambos.

5) Utilización de una función heurística para guiar la búsqueda.

Se tienen dos formas de incorporar la información heurística en el proceso de búsqueda basado en reglas:

- *Matching* indexado.

  En vez de buscar en todo el conjunto de reglas, éstas se pueden indexar por medio del estado actual del problema.

- *Matching* con variables.

  Cuando se utiliza un *Matching* no literal, no sólo es importante reconocer el *Match* entre el patrón de la regla y el estado del problema, sino también el saber qué asignaciones de variables fueron hechas justo en el momento en el que ocurrió el *Match*.

- *Matching* por aproximación.

  Cuando hay reglas que contienen precondiciones no explícitas en algún estado del problema, se requieren de mecanismos de *Matching* que permitan inferir precondiciones.
  Este *Matching* se vuelve más complejo cuando se tienen aproximaciones a las precondiciones de las reglas, sobre todo cuando se trabaja con descripciones físicas del mundo que nos rodea (la regla más parecida).

El resultado del proceso de *Matching* es una lista de las reglas cuyas partes izquierdas coinciden con el estado actual del problema. El método de búsqueda puede entonces escoger entre todas la regla específica y el orden en que realmente se aplicará.

La manera en la que interactúan el proceso de *Matching* y el de búsqueda depende de la estructura de ambos.

5) Utilización de una función heurística para guiar la búsqueda.

Se tienen dos formas de incorporar la información heurística en el proceso de búsqueda basado en reglas:

- Incluyéndola dentro de las mismas reglas.

- Como una función heurística que evalúa estados individuales del problema y determina que tan adecuados son.

Una función heurística traduce de una descripción del estado del problema a una medida, generalmente numérica, de qué tan aceptable es ese estado en especial.

El propósito de las funciones heurísticas es el de guiar al proceso de búsqueda hacia la dirección más prometedora como solución, sugiriendo qué ruta escoger entre todas las posibles. Entre mas acertada sea la función para estimar los verdaderos méritos de cada nodo en el árbol o gráfica de búsqueda, más directo será el proceso de solución del problema. En el extremo que la función sea perfecta, no se requeriría de un proceso de búsqueda, pues la función heurística proporcionaría una solución directa

En la mayoría de los problemas no es fácil la definicion de funciones heurísticas, pues sólo se obtienen aproximaciones adecuadas para ciertas aplicaciones.

A continuación se muestran algunas de las estrategias de control, de propósito general, comúnmente conocidas como *Weak Methods* (métodos débiles), llamados así porque no soportan la explosión de combinaciones que presentan algunos problemas (los métodos de búsqueda son **generalmente vulnerables a la explosión de combinaciones).**

Aunque se ha observado la limitada eficacia de los métodos para resolver problemas difíciles por sí mismos, éste ha sido uno de los resultados más importantes que surgieron de las dos últimas décadas en investigación sobre Inteligencia Artificial y continúan formando el centro en la mayoría de los sistemas inteligentes en combinación con otros métodos La correcta elección de alguna técnica en especial, dependerá de las características del problema específico cuyo análisis se mencionó anteriormente.

6.5.1 *Generate and test*

Esta estrategia de control es la más simple de todas, la cual consiste de los siguientes pasos:

1) Genera una posible solución.

Ya sea generar un punto particular en el espacio del problema o bien, generar una ruta solución a partir de un estado inicial.

2) Probar si ésta es realmente una solución, comparando el punto generado con el conjunto de metas o soluciones aceptables.

3) Si es una solución viable terminar, de lo contrario regresar al punto 1.

Si la solución existe eventualmente se encontrará con este método. Este algoritmo es un procedimiento de búsqueda *Depth-first*, dado que debe darse una solución completa antes de que pueda ser probado. En forma sistemática se reduce a hacer una búsqueda exhaustiva en el espacio de estados. También se tiene la posibilidad de generar soluciones aleatoriamente con su correspondiente prueba, pero esto no garatiza que se llege a una solución.

La forma más directa y segura de implementar este método es con un árbol de búsqueda *depth-first* con evaluación y retroceso (*Backtracking*).

Como método aislado no presenta ventajas significativas, pero al utilizarlo en combinación con otros métodos se pueden sobrellevar las limitaciones y crear uno más efectivo.

6.5.2 *Breath-first-search.*

En este procedimiento todos los nodos en cada nivel del árbol son examinados antes de pasar al siguiente nivel. Un procedimiento de búsqueda *breadth-first* garantiza el encontrar una solución si ésta existe, considerando que hay un número finito de ramas en el árbol. Si existe solución, entonces existe una ruta de longitud finita y el procedimiento es el siguiente:

- Revisar todas las rutas de longitud uno.
- Revisar todas las rutas de longitud dos y así sucesivamente hasta encontrar el nodo o estado solución y como consecuencia la solución que tiene la ruta más corta.

Este procedimiento de búsqueda tiene los siguientes inconvenientes:

- Requiere mucha memoria.
- Realiza mucho procesamiento, aún para soluciones cortas.
- Incremento notable del número de nodos si se utilizan operadores redundantes o irrelevantes.

Por estas razones el método es particularmente inapropiado en la búsqueda de soluciones que tengan un número excesivo de rutas de longitud superior a dos nodos.

6.5.3 *Best-first-search.*

En este método se busca combinar las ventajas de los métodos *depth-first* y *breath-first* en uno solo. En cada paso de este procedimiento de búsqueda se selecciona el nodo que sea más adecuado, medido por una función heurística apropiada para cada uno de los nodos analizados.

Después de seleccionar el nodo más adecuado, según la función, se toma como nuevo punto inicial para continuar el proceso. Cuando se concluye que no se llegó a una solución porque se encuentra un nodo menos adecuado en ese nivel, se regresa al nivel superior y se selecciona el siguiente nodo que en orden de prioridad continúe del antes seleccionado, iniciando nuevamente el proceso.

Toda la información de los estados analizados, en el transcurso del proceso, es almacenada para obtener la información de la ruta de solución, además de los resultados de las funciones heurísticas para considerar el siguiente nodo de búsqueda, llegando al final del proceso al encontrar el nodo que satisfaga los requerimientos del problema.

Un método gráfico es muy útil como auxiliar pues facilita el seguimento manual del método.

6.6 Representación del Conocimiento.

Para resolver problemas complejos de Inteligencia Artificial, se requiere de una base amplia de conocimiento y mecanismos de manejo para encontrar soluciones adecuadas.

Hasta ahora se han visto mecanismos generales de manejo del conocimiento utilizando búsquedas. Estos métodos son muy generales y por su generalidad son limitados. Existe una variedad de formas para representar el conocimiento (hechos) que han sido explotadas por sistemas de Inteligencia Artificial. Antes de explicar algunas formas de representación del conocimiento, revisemos dos entidades involucradas en dicha representación:

- Hechos: Verdades en palabras relevantes (estas son las cosas que se quieren representar)
- Representación de hechos utilizando algún formalismo elegido (estas son las cosas que realmente se pueden manejar)

Deben existir funciones que traduzcan de los hechos a su representación y viceversa. Una representación de hechos común para todos y que merece especial atención, es la utilización del lenguaje natural.

6.6.1 Representación de hechos simples en lógica.

Una manera particular de representar hechos es el lenguaje de la lógica.

Partiendo del lenguaje natural, a través de la lógica obtenemos una representación formal de hechos. Por ejemplo, tenemos la siguiente proposición lógica:

Rufus es un perro	**Perro (Rufus)**
(Lenguaje Natural)	*(Formalismo en lógica de proposición)*

El formalismo de la lógica es atractivo por la facilidad que presenta para derivar nuevos conocimientos a partir de cierto conocimiento previo: Deducción Matemática. Con este formalismo se puede derivar un nuevo hecho verdadero, que proviene lógicamente de otros hechos previamente conocidos. Esta capacidad de deducción se observa como una manera de obtener respuestas a preguntas y soluciones a problemas. La utilización de la representación del conocimiento en lógica permite solucionar problemas.

Lógica de predicado.

Podemos representar hechos en forma de proposiciones lógicas:

Está lloviendo *LLOVIENDO*

Cuando llueve no hay sol *LLOVIENDO* ——→ *NO SOL*

Sin embargo, las proposiciones lógicas se ven limitadas a representar hechos muy simples.

Existe otro formalismo conocido como lógica de predicado, que permite representar una mayor gama de hechos. La lógica de predicado es un buen medio de representación del conocimiento y provee una manera de deducir nuevos hechos.

Los mecanismos de deducción utilizando representación en lógica de predicado se pueden observar al tratar de probar un teorema propuesto (solución del problema) utlizando ciertos axiomas (conocimiento previo). Puede darse el caso que este mecanismo de deducción no llegue a demostrar el teorema (solución) propuesto. A continuación se muestran ejemplos de representación de hechos en forma de lógica de predicado:

Marco fue un hombre ——→ hombre (Marco)

Esta representación captura el hecho crítico de "ser hombre" y omite la idea del tiempo (fue). Esta omisión es o no aceptable, dependiendo del uso que se le pretende dar al conocimiento.

Marco era de Pompeya ——→ Pompeya (Marco)

Todos los hombres de Pompeya eran Romanos

$\forall$ X Pompeya (X) $\therefore$ Romano (X)

Todos los Romanos eran leales a César o lo odiaban

$\forall$ X Romano (X) $\therefore$ leal (X,César) ó odia (X,César)

El proceso de convertir oraciones de lenguaje natural a fórmulas en lógica de predicado puede ser un proceso no difícil, por lo siguiente:

1) Algunas oraciones pueden ser ambiguas.

2) La elección de los hechos críticos a representar, dependerá del uso que se le dé al conocimiento.

3) En ocasiones, las oraciones no contienen toda la información necesaria para razonar sobre cierto tema. Entonces es necesario que contengan la información que es considerada obvia para la gente, pero que es estricamente necesaria para poder deducir mediante los hechos representados.

Uno de los mecanismos de deducción, utilizando representación en lógica de predicado, es el conocido como Resolución.

Resolución, es un procedimiento que permite razonar con declaraciones en lógica de predicado y parte de la idea de demostrar que la negación de un hecho produce una contradicción con los hechos ya conocidos.

Una de las dificultades que presenta la representación de hechos en lógica de predicado es la existencia de problemas con cierto tipo de información que no es representable por medio de esta técnica. Nuevamente llegamos al punto de dependencia entre el problema específico y la elección de una técnica adecuada de solución como se muestra en los siguientes ejemplos no representables en lógica de predicado:

El dia de hoy, hace mucho calor

No podemos representar grados relativos de calor, ¿qué tanto es mucho?.

Si no hay evidencia de lo contrario, asume que cada adulto que conoces sabe leer

No podemos representar un hecho que debemos inferir a partir de la ausencia de otros.

Es mejor tener más piezas en el tablero de ajedrez, que el número de piezas que nuestro oponente tiene

No podemos representar información hurística.

Existen varias técnicas de representación del conocimiento que tratan de enfocar estos problemas como son:

- Razonamiento no monótono (*Nonmonotic logic*)
- Razonamiento probabilístico.

6.6.2 Razonamiento no monótono.

Los sistema basados en lógica de predicado son monótonos en el sentido que el número de declaraciones (hechos) conocidos como verdaderos, estrictamente crece con el tiempo. Nuevos hechos se agregan a la base de conocimientos por los mecanismos de deducción. Cuando no se puede probar la veracidad de un hecho, éste simplemente es descartado.

El trabajar con un sistema "monótono" en este sentido tiene algunas ventajas, como son:

- Cuando agregamos un nuevo hecho no es necesario verificar su consistencia pues proviene lógicamente del conocimiento previo.
- No es necesario recordar la serie de hechos utilizados en la deducción de un nuevo hecho, porque no hay riesgo de que alguno de estos hechos desaparezca de la base de conocimiento.

Sin embargo, estos sistemas limitan la representación en dominios de problemas que presentan:

- Información incompleta
- Situaciones cambiantes
- Generación de hechos que se asumen

Cuando la información que se maneja está incompleta, se pueden adivinar ciertos hechos mientras no exista una evidencia de contradicción. Esta forma de adivinar hechos a partir de la ausencia de otros se conoce como Razonamiento por Omisión. Por ejemplo si alguien nos invita a cenar y en el camino pasamos por una florería, pensamos que sería buen detalle obsequiar un ramo de flores al anfitrión. Nosotros no sabemos precisamente si al anfitrión le gustan las flores, pero intuitivamente utilizamos una regla general:

Dado que a la mayoría de la gente le gustan las flores, al anfitrión le gustarán, a menos que él haya evidenciado lo contrario.

Este tipo de razonamiento es no monótono porque los hechos que se derivan provienen de la falta de información del resto de los hechos.

Una técnica común de razonamiento por omisión es la conocida como *la elección más probable* (*the most probable choice*). Si sabemos que un hecho, dentro de un conjunto, debe ser verdadero, en la ausencia de información, escogemos el que más se le parezca.

Una descripción precisa del razonamiento por omisión, debe relacionar la falta de información X con una conclusión Y, por ejemplo:

Definición 1. Si X no se conoce, entonces concluye Y.

Definición 2. Si X no se puede probar, entonces concluye Y.

Definición 3. Si X no se puede probar en cierto tiempo, entonces concluye Y.

6.6.3 Razonamiento probabilístico.

Hasta ahora, se han definido hechos que son ciertos o bien falsos, o bien hechos que desconocemos. Hay que considerar ahora la posibilidad de tener un hecho que probablemente sea cierto.

Hay tres tipos de situaciones que se prestan para utilizar razonamiento probabilístico:

a) El mundo relevante del problema es realmente aleatorio, por ejemplo:

   - La distribución de electrones en un átomo.

b) El mundo relevante del problema no es aleatorio si se cuenta con información suficiente, pero es difícil obtenerla.

   - Probabilidad de cura bajo cierto tratamiento de un paciente en particular.

c) El mundo del problema aparenta ser aleatorio porque no se tiene descrito en un nivel adecuado, por ejemplo:

   - El reconocimiento de patrones de voz.

En este caso la teoría matemática de probabilidad provee una manera de describir y manejar conocimiento o hechos inciertos.

6.7 Herramientas para la implantación de Sistemas Inteligentes.

Como se mencionó, el estudio de los problemas de Inteligencia Artificial ha permitido el desarrollo de nuevas técnicas de programación que se basan en el procesamiento simbólico, mecanismos de deducción, así como en la representación y el manejo del conocimiento.

Se han discutido algunas de las características que presentan los problemas y las diversas técnicas de solución que se pueden aplicar. Ahora se mencionarán algunas de las herramientas de programación con que se cuenta en la actualidad para implantar sistemas inteligentes.

Las características deseables en cualquier lenguaje de programación para solucionar problemas, incluyendo aquellos enfocados a la Inteligencia Artificial se resumen en los siguientes puntos:

- Una variedad de *data types* para poder describir todos los tipos de datos que un sistema de información complejo necesita.
- Habilidad para descomponer el sistema en unidades más pequeñas y que sea razonablemente fácil efectuar modificaciones (modularidad).
- Control flexible sobre las estructuras, que facilite recursión y descomposición paralela del sistema.
- Habilidad para comunicarse interactivamente con el sistema, tanto para desarrollo, como para el uso efectivo de sistemas finales.
- Habilidad para producir código eficiente para que el rendimiento (*performance*) sea aceptable.

Es interesante observar que aunque algunas de estas características se consideran importantes ahora en cualquier sistema de información razonablemente complejo, algunas de ellas fueron requeridas inicialmente para sistemas de Inteligencia Artificial[K].

Se ha mencionado la gran dependencia que exite entre el problema y la estrategia de solución. Los diversos problemas de Inteligencia Artificial determinan las características deseables de las nuevas técnicas de programación, siendo las más generales:

- Facilidad para el manejo de listas, ya que éstas son la estructura más utilizada en casi todos los programas de Inteligencia Artificial.

- Transparencia referencial, es decir, si para alguna variable no se ha asignado valor en el programa, no hay la necesidad de almacenar el nombre de dicha variable, sino hasta que ésta adquiere un valor.

- Facilidad para evaluar la coincidencia de patrones (*Pattern Matching*), tanto para identificar tipos de datos como para el control. La coincidencia de datos es una parte importante en la utilización de una base de conocimientos compleja, mientras que la coincidencia para el control forma las bases de la ejecución de un sistema en producción.

- Facilidad para realizar algún tipo de deducción automática y para almacenar en alguna base de datos el resultado de dichas deducciones.

- Facilidad para construir estructuras complejas de conocimiento.

- Mecanismos por medio de los cuales el programador pueda agregar conocimiento adicional.

- Estructuras de control que faciliten un ambiente orientado a metas.

No existe ningún lenguaje de progamación que contenga todas las características anteriores. Algunos lenguajes cumplen mejor algunas características que otros. A continuación se presenta una breve descripción

de los lenguajes que proporcionan algunas de las características mencionadas.

a) IPL (*Information Processing Language*): Desarrollado por Newell en 1960. Fue el primer lenguaje que introdujo el procesamiento de listas. Marca el inicio del desarrollo de lenguajes de programación enfocados a la Inteligencia Artificial.

b) LISP (*LISt Processing*): Es uno de los lenguajes más utilizados en Inteligencia Artificial y su estructura principal es la lista.

c) INTERLISP: Es un dialecto de LISP.

d) SAIL: Es una derivación de ALGOL con algunas características adicionales, incluyendo el soporte primitivo de memoria asociativa (direccionamiento por contenido). De todos los lenguajes de Inteligencia Artificial, SAIL es el más similar a los lenguajes tradicionales de propósito general.

e) PLANNER: Diseñado para representar características tradicionales, así como un ambiente análogo a metas.

f) KRL (*Knowledge Representation Language*): Lenguaje que soporta estructuras complejas del conocimiento.

g) PROLOG (*PROgramming in LOGic*): Lenguaje basado en hechos y reglas.

El diagrama genealógico (figura 25) muestra la evolución de estos lenguajes, cada uno con características específicas, que dependiendo del tipo de problema a resolver, algunos son más convenientes que otros.

Las nuevas técnicas de programación han tenido gran impacto sobre la programación tradicional de propósito general, no sólo en aplicaciones de Inteligencia Artificial sino en aplicaciones de cualquier tipo.

La programación tradicional, desde el lenguaje binario hasta los lenguajes de alto nivel como BASIC o PASCAL, se caracteriza porque el programador debe describir completamente la manera en que se desean obtener resultados en vez de describir lo que se desea obtener como resultado. A esta familia de lenguajes, como BASIC, PASCAL o ADA, se les conoce como lenguajes imperativos.

Los lenguajes imperativos están constituidos de comandos que especifican paso a paso las acciones a realizar, explícitamente detallan el flujo de control necesario para obtener un resultado, esto es, están orientados a la máquina en la que se ejecutan.

Los lenguajes declarativos, al contrario de los imperativos, intentan separar la tarea a realizar de la manera en que la computadora la realizará. Los lenguajes declarativos permiten describir un conjunto de relaciones o funciones a evaluar. La ejecución de un programa declarativo, se reduce al uso de las definiciones para obtener los resultados deseados. La forma en que se obtendrán dichos resultados es tarea del lenguaje que se utilice.

Los lenguajes LISP y PROLOG son ejemplos de lenguajes declarativos. PROLOG tiene características adecuadas a los requerimientos del sistema Normalizador, objeto de este trabajo.

PROLOG remonta sus orígenes a principios de los años setentas, desarrollado por Alain Colmerauer y Phillippe Roussel en Marsella, [H] Francia, como una implantación física de la programación en lógica.

La programación en lógica surge por los esfuerzos realizados alrededor de los años cincuentas para automatizar la demostración de teoremas matemáticos a través de una computadora.

En PROLOG se representa el conocimiento mediante una variante de lógica de predicado, con oraciones que toman la forma:

Conclusión *Si* *Condiciones*

Este tipo de oraciones son conocidas como cláusulas de Horn, en honor al logista Alfred Horn, quien estudió sus propiedades lógicas, por ejemplo:

Diagrama genealogico de lenguajes de Inteligencia Artificial.

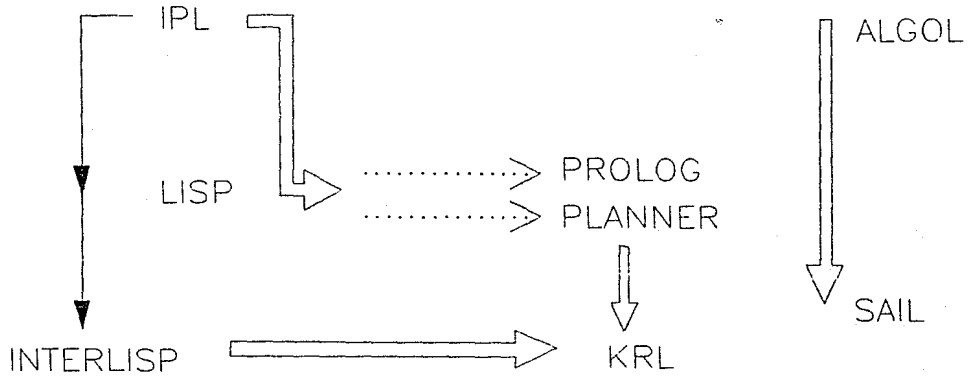
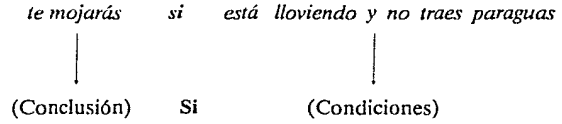


Fig. 25

Fuente : Rich, Elaine "Artificial ..."



Por otro lado, utiliza el mecanismo de deducción de razonamiento regresivo, ya que éste es lo suficientemente mecánico como para implementarse en una computadora.

Los mecanismos de *matching* utilizados por PROLOG están dados por el algoritmo de unificación, una técnica de *matching* con variables.

PROLOG es un caso práctico de la aplicación de varias técnicas de Inteligencia Artificial, dando como resultado una herramienta importante para implantación de Sistemas Inteligentes.

II
Paquete Normalizador

1 Desarrollo.

1.1 Utilidad de un paquete normalizador.

Aún cuando se reconoce la importancia de la normalización como uno de los pasos dentro del diseño de una base de datos, son pocas las personas que realmente comprenden y usan los algoritmos de normalización en la práctica[E].

Descuidar este paso del diseño puede traer serias complicaciones cuando el sistema se encuentra en producción, en el momento que se necesiten hacer cambios a la base de datos o al momento de actualizarla.

La utilización de estos algoritmos asegura que no se pasa por alto ninguno de los puntos en el proceso de normalización.

Resulta casi imposible que todos aquellos que se dedican al diseño de bases de datos dominen perfectamente los algoritmos. En la medida que una base de datos aumenta de tamaño, su normalización se vuelve más complicada, a tal grado que el consumo de recursos puede llegar a límites inaceptables. Entiéndase por tamaño de una base de datos al número de atributos y de dependencias funcionales y no al volumen de información.

Es por todo esto que resulta de gran utilidad contar con un paquete que le facilite al diseñador la tarea de la normalización.

El usuario de un paquete de esta naturaleza no necesita ser un experto en normalización para obtener los resultados deseados en poco tiempo. Basta con que entienda suficientemente bien el problema, que conozca los atributos a manejar y las dependencias entre ellos.

Por otro lado, se puede realizar este importante paso en el proceso del diseño de una base de datos en un equipo diferente del que se utilice para la implantación del sistema.

1.2 PROLOG y el enfoque relacional.

El enfoque de base de datos relacional se ha caracterizado porque introduce terminología matemática, de hecho, se fundamenta en la teoría matemática de relaciones.

Los algoritmos de normalización manejan conceptos propios de la teoría matemática de relaciones tales como dependencias funcionales entre conjuntos de atributos. Estos algoritmos se pueden concebir entonces como una secuencia lógica de pasos para demostrar la existencia o ausencia de ciertos tipos de dependencias entre atributos.

Por otro lado se mencionó que PROLOG tuvo sus orígenes en la programación en lógica, que surgió para la demostración automática de teoremas. PROLOG por tanto, es un lenguaje poderoso en el manejo de proposiciones lógicas y de objetos que guardan cierta relación entre sí.

Los algoritmos de normalización no constituyen una secuencia de pasos desde el punto de vista procedural, sin embargo, forman una secuencia de verificaciones lógicas para obtener tal o cual conclusión sobre el tipo de dependencias y las acciones a tomar en tales casos. Es aquí donde encontramos una liga conveniente entre los algoritmos de normalización y PROLOG.

Esta liga matemática entre la teoría relacional y PROLOG hace a este último una herramienta ideal para implementar los algoritmos de normalización de relaciones[G].

Los conceptos en los que dichos algoritmos se apoyan son fácilmente representables. Estos se reducen a la verificación de ciertas condiciones, como el tipo de dependencias entre conjuntos de atributos y el razonamiento regresivo de PROLOG, lo que facilita su implementación.

Además de estos aspectos importantes existen en el mercado una gran variedad de intérpretes y compiladores de PROLOG lo cual lo hace accesible y versátil. Específicamente se utiliza Turbo PROLOG para el desarrollo del paquete normalizador, por la facilidad de adquisición y transportabilidad de las computadoras personales.

A continuación se presenta formalmente la estructura y las características de los componentes de un programa en Turbo PROLOG.

Un programa en Turbo PROLOG consiste en una descripción del problema que se pretende resolver. Esta descripción está constituida por tres componentes:

1) Nombres y estructuras de objetos involucrados en el problema.

2) Nombres de las relaciones que existen entre los objetos.

3) Hechos y reglas describiendo estas relaciones.

La descripción dentro de un programa se utiliza para especificar la relación que se desea obtener entre los datos de entrada y los datos que se generan a la salida. Esta descripción consta de una lista de oraciones lógicas que se pueden representar en forma de hechos:

ESTA LLOVIENDO

o en forma de reglas:

TE MOJARAS SI ESTA LLOVIENDO Y NO TIENES PARAGUAS

Turbo PROLOG permite deducir hechos a partir de reglas y hechos previos:

A ELENA LE GUSTA EL TENIS

A JUAN LE GUSTA EL FUTBOL

A ELENA LE GUSTA X SI A JUAN LE GUSTA X

deducimos entonces que:

A ELENA LE GUSTA EL FUTBOL.

La filosofía de Turbo PROLOG está orientada a metas, es decir, una vez descritas las relaciones entre objetos involucrados a través de hechos y reglas, se proporciona una meta por alcanzar y Turbo PROLOG hará uso de los hechos y reglas para tratar de alcanzar la meta proporcionada. Existen dos modalidades para establecer una meta :

1) De manera interactiva, donde la meta por alcanzar es proporcionada por el usuario, o

2) Formando parte del código del programa *Hard Coded.*

Al tratar de alcanzar una meta, Turbo PROLOG utiliza un mecanismo de *backtracking*, el cual una vez obtenida alguna solución, reestablece las condiciones iniciales y las evalúa nuevamente para verificar si existen otros valores de variables que den lugar a nuevas soluciones. Así, Turbo PROLOG obtiene todas las posibles soluciones a alguna meta dada.

Turbo PROLOG es un lenguaje de alto nivel que ofrece una sintaxis muy sencilla, fácil de aprender y que típicamente utiliza diez veces menos líneas de programa que otros lenguajes tradicionales de programación[D].

Como se mencionó anteriormente, un programa en Turbo PROLOG es una descripción constituída por nombres de objetos, relaciones, hechos y reglas que describen a las relaciones en sí.

La estructura de un programa en Turbo PROLOG para poder realizar dichas descripciones es la siguiente:

- *DOMAINS*
- *PREDICATES*
- *GOAL* (OPCIONAL)
- *CLAUSES*

Los objetos involucrados en el programa pertenecen a cierto dominio y deben ser especificados en la sección *DOMAINS.*

Esta sección consta de :

DOMAINS

nombre objeto 1 = dominio1

nombre objeto 2 = dominio2

. . .

. . .

nombre objeto n = dominion

Para dar nombre a los diversos objetos tenemos la siguiente regla:

Nombres de Objetos: Deben comenzar con letra minúscula, seguida de cualquier secuencia de carácteres (letras números y el carácter de subrayado '_').

Los dominios dominio1, dominio2, etc. pueden ser cualquiera de los siguientes dominios estándares de Turbo PROLOG :

char:	Cualquier carácter encerrado entre apóstrofos, por ejemplo: '%'.
integer:	Enteros (del -32768 al 32767).
real:	Reales, con signo, punto decimal y exponente opcionales, por ejemplo:7, 7.908, -3.8E12
string:	Cualquier secuencia de carácteres encerrados entre comillas.
symbol:	Se tienen dos formatos :

1) Secuencia de letras, números y carácter de subrayado, con la primera letra minúscula.

o bien,

2) Cualquier secuencia de carácteres encerrada entre comillas (equivalente a un string).

dbasedom:	Cualquier átomo perteneciente a la base de datos del programa en Turbo PROLOG.

Cualquier otro creado por el usuario.

En la sección *PREDICATES* se definen todas las relaciones del programa, así como los objetos involucrados en cada una de las relaciones :

PREDICATES

nombre de la relación_1(obj1,obj2,...,objn)

nombre de la relación_2(obj1,obj2,...,objk)

. . . .

nombre de la relación_n(obj1,obj2,...,obji)

Los nombres de las relaciones se forman con cualquier combinación de letras (mayúsculas o minúsculas), números o carácteres. Los objetos involucrados pueden ser cualquiera de los declarados en la sección de *DOMAINS*.

Los hechos y reglas que describen las relaciones, se especifican en la sección *CLAUSES*.

En Turbo PROLOG, un hecho consiste en una relación que afecta a uno o varios objetos y tienen la siguiente forma general:

Hechos:

nombre_relación (obj1,obj2,...,objn).

Las reglas toman la siguiente forma general :

Reglas:

relación_y(obj1,..,objm) *if* relación_a(obj,..,obj)
and relación_b(obj,..,obj)
and . . . .
. . . .
and relación_x(obj,..obj).

Los hechos son aseveraciones sobre algo y al aparecer éstos dentro de la sección *CLAUSES*, son cargados a la base de datos del programa, para poder

ser consultados. Es por ésto que PROLOG tiene la reputación de estar enfocado a consultas, al hacer uso de su propia base de datos.

Tenemos por ejemplo las siguientes oraciones en español:

- A julio le gusta el volibol
- A martha le gusta leer
- A martha le gusta X si a julio le gusta X

Podemos describir la relación le_gusta entre los objetos persona y actividad de la siguiente manera:

DOMAINS

persona = symbol

actividad = symbol

PREDICATES

le_gusta(persona,actividad)

CLAUSES

le_gusta(julio,volibol)

le_gusta(martha,leer).

le_gusta(martha,X) *if* le_gusta(julio,X).

Podemos distinguir los objetos involucrados y sus respectivos dominios (persona y actividad), así como la relación que existe entre ambos (relación le_gusta(persona,actividad)). En la sección *CLAUSES*, se observan hechos le_gusta(julio,volibol) y le_gusta(martha,leer) y una regla (le_gusta(martha,X) if le_gusta(julio,X)). Podemos ejecutar este programa especificando una meta por a alcanzar. Si ésta se desea proporcionar de manera interactiva, únicamente hay que ejecutar este programa y aparecerá el siguiente *prompt*:

```
› GOAL:
```

Inmediatamente después, podemos definir la meta que deseamos alcanzar, por ejemplo :

```
› GOAL:

› le_gusta(martha,leer)
```

Turbo PROLOG consulta los hechos y reglas del programa para determinar si la meta proporcionada ha sido o no alcanzada. En este caso, la meta se alcanzó al encontrar el hecho le_gusta(martha,leer) y Turbo PROLOG respondió con *True*.

Así como en la regla del programa ejemplo se utilizó la variable X, también dentro de las metas se pueden incluir variables, como por ejemplo :

```
› GOAL:

› le_gusta(martha,X)
```

Y Turbo PROLOG responde:

```
› X = leer

› X = volibol

› two solutions.
```

Para esta meta, Turbo PROLOG consulta los hechos y reglas del programa para obtener todas las soluciones, la primera obtenida a partir del hecho le_gusta(martha,leer), y la segunda obtenida a partir de la regla le_gusta(martha,X) if le_gusta(julio,X).

Estas metas, como se mencionó anteriormente, se pueden proporcionar dentro del programa a través de la sección *GOAL*, que es opcional, pues precisamente se permite ejecutarlas en la modalidad interactiva o bien en modalidad *Stand-Alone*.

Las variables forman una parte importante en la programación en Turbo PROLOG. A diferencia de otros lenguajes, Turbo PROLOG maneja el concepto de Variables Libres y Variables Acotadas (*bound variables*). Una

Variable Libre es aquella de la cual Turbo PROLOG desconoce su valor. Una Variable acotada es aquella a la que Turbo PROLOG le ha asignado un cierto valor y se dice que esta sujeta a ese valor. La asignación de valores o desasignación de valores a variables esta involucrada con el mecanismo que Turbo PROLOG utiliza para dar solución a una meta.

Para encontrar una solución a una meta, Turbo PROLOG busca encontrar una coincidencia entre la meta y alguna cláusula de la sección *CLAUSES*; al proceso que se encarga de encontrar esta coincidencia se le conoce como Proceso de Unificación y su total comprensión es vital para la programación en Turbo PROLOG.

Una meta puede estar compuesta por varias submetas por alcanzar. Para que una meta compuesta tenga éxito, es necesario que todas las submetas también lo tengan.

Al tratar de dar solución a alguna meta, se pretende obtener una coincidencia de la meta con alguna cláusula de la sección *CLAUSES*. Puede ser un hecho, o bien, una regla.

Las variables juegan un papel muy importante, pues de ellas depende el control para dar solución a alguna meta. Para dar nombre a las variables utilizamos la siguiente regla :

- **Nombres de Variables:** Comenzar por letra mayúscula, seguida de cualquier secuencia de carácteres.

Para nuestro ejemplo, la meta le_gusta(martha,X), hace uso de la variable X.

En el momento de iniciar la búsqueda de solución, X es libre. Se pasa por el primer hecho de la base de datos del programa: le_gusta(julio,volibol) y observamos que no existe coincidencia, dado que martha no es una variable y no coincide con julio. Turbo PROLOG intenta con el siguiente hecho : le_gusta(martha,leer). Aquí, se encuentra una coincidencia entre la meta y la cláusula, la variable X toma inmediatamente al valor leer. Una vez obtenida esta solución, Turbo PROLOG ejectua el *backtracking* para verificar si existe otra solución.

Al efectuar el *backtracking,* Turbo PROLOG encuentra una coincidencia entre la meta y la cláusula le_gusta(martha,X) if le_gusta(julio,X). En este momento la variable X es libre y ahora se debe tratar de satisfacer el lado

derecho de la regla: le gusta(julio,X) obteniéndose que se satisface cuando X = volibol, y por tanto este valor se le asigna a la variable del lado izquierdo de la regla, obteniéndose así la segunda y última solución.

Para controlar la asignación de valores, según las circunstancias, se hace uso del Algoritmo de Unificación:

Algoritmo de Unificación.

Las variables libres se unifican (pueden coincidir) con cualquier término. En ese momento, la variable queda acotada con el valor del término al que se unificó.

Una constante sólo se puede unificar a ella misma, o bien, a una variable libre.

Proceso para evaluación de metas (Proceso de Unificación):

1) Si una meta se descompone en submetas, las submetas se deben de satisfacer de izquierda a derecha.

2) Las cláusulas de predicado deben de probarse en el orden en que se escribieron en el programa.

3) Cuando una submenta coincide con la parte izquierda de una regla, la parte derecha de esa regla debe satisfacerse enseguida. (La parte derecha constituye un nuevo conjunto de submetas).

Estos son los conceptos básicos de Trubo PROLOG, sin embargo, existen otras características igualmente importantes en la programación en Turbo PROLOG, las cuales se mencionan brevemente a continuación:

Recursividad

La recursividad es una técnica de programación importante para Turbo PROLOG, ésta es utilizada normalmente en dos situaciones:

1) Cuando las relaciones son descritas con la ayuda de ellas mismas y

2) Cuando objetos compuestos son parte de otros objetos compuestos, ésto es, objetos recursivos. El utilizar objetos recursivos permite definir objetos en donde no se sabe con anterioridad el número de elementos que lo formarán.

Listas

Las listas son parte de la estructura básica de los programas en Turbo PROLOG, éstas por su fácil representación, se han convertido en estructuras de uso común. Los elementos dentro de ellas se separan por comas "," y encerrados entre corchetes; los elementos que se encuentren dentro de una misma lista deben pertenecer al mismo dominio.

Al momento de procesar la lista se divide en dos partes, éstas son: *head* y *tail*, encabezado y resto, es decir, la primera parte o encabezado contiene al primer elemento de la lista y la segunda parte o resto contiene como su nombre lo indica, el resto de los elementos.

Las listas son la herramienta más poderosa en los lenguajes descriptivos como Turbo PROLOG.

Es así que Turbo PROLOG ofrece las características necesarias para implantar programas que traten de dar solución a problemas de inteligencia artificial y en el presente caso para la implantación del paquete normalizador.

1.3 Descripción del Paquete Normalizador.

El sistema normalizador es una herramienta aplicable a una parte del diseño de bases de datos que como ya se mencionó persigue la eliminación de anomalías al momento de acceso a la misma.

El mecanismo de operación es muy sencillo, está dirigido no sólo a especialistas en sistemas sino también a usuarios finales.

El usuario que utilice este paquete tiene la facilidad de realizar diversas pruebas, es decir, cambiar la semántica de los atributos a normalizar y con ésto, lograr resultados más adecuados a la realidad que los gobierna.

El paquete normalizador se ha elaborado conscientemente de las limitaciones téoricas y prácticas que tiene. Estas consideraciones provocaron que algunos conceptos no se hayan incluido como característica de este paquete. Existen varias razones por las cuáles se hizo de la forma y cobertura que actualmente tiene. Una de las más importantes es:

La teoría de normalización contempla como nivel máximo la quinta forma normal (5FN), para la cual no existen, en la actualidad, algoritmos que la puedan soportar a nivel automatizado, por lo que el normalizador se limitó a cubrir hasta la tercera forma normal (3FN) que sí esta respaldada con algoritmos prácticos y la cual consideramos como etapa aceptable dentro del proceso de normalización.

Para esta determinación se tomaron en cuenta las dificultades de la implantación a una etapa superior (FNBC, 4FN, 5FN) contra los beneficios que brindan y se concluyó que era demasiado esfuerzo para muy escasos beneficios, este esfuerzo se compone de:

- Desarrollo de algoritmos que cubrieran el normalizador en 4FN y 5FN. Aunque existen algoritmos para llegar a FNBC, tampoco brindan los suficientes beneficios para implantarlos. Los algoritmos para 4FN y 5FN serían extremadamente más complicados que los ya mencionados, dada la necesidad de realizar un análisis profundo de aspectos de semántica en la relación específica, asi como instancias mismas de los datos que se quieran normalizar. Los beneficios que se obtendrían, suponiendo que se tuvieran los algoritmos adecuados serían

mínimos ya que estas formas normales son muy específicas respecto a las circunstancias en que éstas se aplican: casos extremadamente raros.

La operación del sistema se realiza con la interacción del usuario quien deberá introducir el esquema de la relación y las dependencias entre atributos.

Una vez capturada la información el paquete inicia el proceso de normalización, el cual genera como resultado un número mínimo de relaciones indicando la llave y los atributos dependientes.

1.4 Diseño.

La base de conocimientos en la que se apoya el paquete normalizador, contiene los siguientes hechos:

Esquema de la relación.

Se almacena de la siguiente forma:

esquema(nombre_de_la_relación,[atrib1..atribn])

El predicado esquema, contiene el nombre de la relación, seguida por un conjunto ordenado de atributos. Cabe hacer notar que es un conjunto ordenado de atributos, el cual no tiene elementos duplicados y no una lista que podría llegar a tenerlos. Al tener los atributos ordenados disminuimos considerablemente el tiempo de proceso aunque el orden no tiene ningún significado semántico.

Dependencias Funcionales.

Se representan por el siguiente predicado:

depfun(nombre_de_la_relación,[atrib1..atribi], [atribj..atribn])

La primera posición del predicado contiene el nombre de la relación que posee a la dependencia funcional; el primer conjunto de atributos contiene la parte determinante de la dependencia funcional, mientras que el segundo contiene los atributos determinados. Los algoritmos que se muestran posteriormente requieren que las partes determinadas de las dependencias funcionales contengan un sólo atributo. (A la interface de entrada se le han agregado reglas para que realice esta transformación).

Llaves.

Se representan de la siguiente manera:

llave(nombre_de_la_relación[atrib1..atribn])

La primera posición contiene el nombre de la relación y la segunda presenta el conjunto de atributos que forman la llave de dicha relación.

Hechos sobre la descomposición.

Se representan hechos de descomposición, cuando una relación ha sido descompuesta en varias relaciones y se representa:

descomp(nombre_de_la_relación_original, nombre_de_la_relación_final)

La relación original, se descompone en varias relaciones finales y para cada una de ellas se inserta un nuevo hecho de descomposición.

Otro hecho sobre la descomposición es:

en3afn(nombre_de_la_relación)

que permite insertar a la base de datos las relaciones producto de la descomposición que se encuentran en tercera forma normal.

El esquema y las dependencias funcionales constituyen la entrada inicial de datos proporcionada por el usuario a través de la interface desarrollada. Esta interface verifica que todos los atributos que forman una dependencia funcional esten contenidos en el esquema original, a su vez convierte las dependencias funcionales que contienen en su parte determinada más de un atributo en varias dependencias funcionales con un sólo atributo, por ejemplo:

Si la siguiente dependencia funcional es proporcionada;

A,B,C —→ D,E,F

la interface la trasforma en las siguientes dependencias;

A,B,C —→ D

A,B,C —→ E

A,B,C —→ F

Una vez verificada la validez de los datos iniciales, éstos son cargados en la base de conocimiento para su utilización en el proceso de normalización.

1.4.1 Algoritmos de Normalización.

Los datos iniciales, proporcionados por el usuario, pueden contener información redundante que debe ser eliminada, para hacer eficiente el proceso de normalización. Esta redundancia se presenta en dos formas:

- Dependencias funcionales redundantes
- Atributos extraños (superfluos).

Este paquete contiene un proceso inicial que se encarga de analizar los datos de entrada para eliminar la redundancia de información. A este proceso se le ha llamado cobertura no redundante del conjunto inicial de datos.

1.4.1.1 Cobertura no redundante del conjunto inicial de datos.

Este proceso tiene como objetivo transformar el conjunto inicial de datos en otro conjunto que contenga la misma información potencial pero que sea mínimo. Puede darse el caso en que el conjunto inicial de datos no contenga ni dependencias funcionales redundates ni atributos extraños, en cuyo caso no habría transformación alguna.

El conjunto de datos que se obtiene como resultado de la cobertura no redundate tiene las siguientes características:

1) Las partes determinadas de las dependencias funcionales constan de un sólo atributo.

Como se mencionó anteriormente, este punto es tarea de la interface del paquete y genera una nueva dependencia funcional por cada atributo contenido en la parte determinada.

Aparentemente se agregan dependencias funcionales, hecho que puede parecer una contradicción al pensar en el objetivo de minimización del conjunto resultante, sin embargo, para efectos de los algoritmos de normalización, es más fácil manejar listas de un sólo elemento y un número mayor de dependencias funcionales.

2) Ninguna parte determinante de las dependencias funcionales contiene atributos extraños.

Un atributo que forma parte del determinante de alguna dependencia funcional, es extraño si no proporciona información para determinar a la parte determinada y por tanto es necesario eliminarlo.

3) Las dependencias funcionales resultantes no son redundantes, es decir, ninguna dependencia funcional puede ser deducida a partir de ninguna otra dependencia funcional.

Cabe mencionar que existen axiomas que permiten deducir nuevas dependencias funcionales a partir de un conjunto inicial de dependencias. Estos son los Axiomas de Armstrong[A] que se presentan a continuación:

REFLEXIVO: Sean A y B conjuntos de atributos de una relación.

Si B es subconjunto de A entonces podemos obtener la dependencia funcional trivial:

$$A \longrightarrow B$$

esto es:

$$\text{Si } B \subseteq A \Rightarrow \quad A \longrightarrow B.$$

TRANSITIVO: Sean A, B y C conjuntos de atributos de una relación y además sabemos que:

$$A \longrightarrow B$$

$$B \longrightarrow C$$

Entonces podemos deducir la nueva dependencia:

$$A \rightarrow C.$$

AUMENTATIVO: Sean A, B y C conjuntos de atributos de una relación. Si sabemos que:

$$A \rightarrow B$$

entonces

$$A, C \rightarrow B, C.$$

Con los Axiomas de Armstrong podemos encontrar todas las dependencias funcionales que se deducen lógicamente del conjunto inicial, obteniendo así el *closure* de un conjunto de dependencias funcionales, que para este caso, no es de utilidad pues se pretende que el conjunto de dependencias funcionales sea mínimo.

Sin embargo, con estos axiomas es posible saber que alguna dependencia funcional proviene lógicamente de otra y por lo tanto, es redundante.

Estas tres características de la cobertura no redundate permiten iniciar el proceso de normalización eficientemente, con un conjunto de dependencias funcionales no redundantes y sin atributos extraños.

Closure de un conjunto X de atributos con respecto a un conjunto de dependencias funcionales.

Para poder explicar los algoritmos que permiten obtener la cobertura no redundante del conjunto inicial de datos, es esencial utilizar el concepto de *closure* de un conjunto X de atributos con respecto a un conjunto de dependencias funcionales, esto es, el conjunto de todos los atributos de A de tal forma que la dependencia $X \rightarrow A$ puede ser deducida a través de los Axiomas de Armstrong.

Una forma más fácil de entender este concepto es: dado un conjunto de dependencias y un conjunto X de atributos, con el *closure* se obtienen todos los atributos que dicho conjunto X determina. Así podemos aplicar el *closure* a todas las partes determinantes de las dependencias funcionales y conocer todos los atributos que cada parte determinante realmente determina.

Algoritmo para la obtención del *closure*.

Sea X un conjunto de atributos, obtenemos el *closure*(X) de la siguiente manera:

Para todas las dependencias funcionales

» Tomar una dependencia funcional (dfi)

» Si la parte determinante de dfi es subconjunto de X:

» Si la parte determinada de dfi no es subconjunto de X:

» X = X U (parte determinada de dfi)

» *CLOSURE(X)*

» De lo contrario

» De lo contrario

» *CLOSURE (X) = X*

Este algoritmo recursivo, encuentra las dependencias funcionales en las que la parte determinante es subconjunto de X y la parte determindada no es subconjunto de X, es decir, que contribuye con nuevos atributos. Finalmente el resultado se obtiene en X.

Eliminación de atributos extraños.

Como se mencionó, una de las características que posee la cobertura no redundante es la ausencia de atributos extraños.

Un atributo es extraño si al eliminarlo de la parte determinante de alguna dependencia ésta se conserva, por ejemplo:

sea la dependencia A, B, C ⟶ D

el atributo A será estraño si se cumple lo siguiente:

$$(A, B, C) - (A) \longrightarrow D$$

quedando

$$B, C \longrightarrow D$$

ya que A no es necesario para determinar a D.

El algoritmo para la eliminación de atributos extraños es aplicable sólo a las dependencias funcionales que contienen más de un atributo en su parte determinante.

Algoritmo para la eliminación de atributos extraños.

Eliminación de atributos extraños (parte determinante)

» Si la parte determinante de dfi tiene más de un atributo

» toma un atributo atri

» elimínalo temporalmente

» obtener el closure de la parte determinante reducida

» Si la parte determinada del dfi es subconjunto del *closure* de la parte determinante reducida

» Elimina el atributo atri definitivamente

» Eliminación de atributos extraños (parte_determinante)

» De lo contrario

» Agregar nuevamente el atributo eliminado

» De lo contrario

Como se puede observar se van eliminando uno a uno los atributos que conforman a la parte determinante de alguna dependencia. Obtenemos el *closure* de la parte determinate reducida. Si la parte determinada de las dependencias en cuestión está contenida en el *closure*, significa que con la parte determinante reducida es suficiente para conocer la parte determinada, por tanto la dependencia no requiere del atributo eliminado, confirmando que es extraño y eliminándolo definitivamente.

Eliminación de dependencias funcionales redundantes.

Similar al algoritmo de eliminación de atributos extraños el algoritmo de eliminación de dependencias funcionales redundantes, elimina temporalmente alguna dependencia funcional. Se obtiene entonces el

closure de la parte determinante de dicha dependencia, pero con respecto al conjunto de dependencias reducido. Si la parte determinada de la dependencia delimitada es subconjunto del *closure*, significa que la dependencia eliminada es redundante, ya que sin ella fue posible deducir la parte determinada y ésta será eliminada definitivamente.

Algoritmo para la eliminación de dependencias funcionales redundantes.

Eliminación de dependencias funcionales:

- » Eliminar temporalmente una dependencia funcional de todo el conjunto
- » Obtener el *closure* de la parte determinante de la dependencia eliminada, con respecto al conjunto de dependencias reducido.
- » Si la parte determinada de la dependencia eliminada es subconjunto del *closure*
  - » Eliminar la dependencia definitivamente
- » De lo contrario
  - » Agregar nuevamente la dependencia funcional eliminada.

Ya con los algoritmos para la eliminación de atributos extraños y dependencias funcionales redundantes, podemos obtener el algoritmo para la cobertura no redundante del conjunto inicial de datos.

- » Para todas las dependencias:
  - » Eliminar todos los atributos extraños(parte determinante)
- » Para todas las dependencias:
  - » Eliminar dependencias funcionales.

1.4.1.2 Descomposición de relaciónes en tercera forma normal.

El algoritmo utilizado para descomponer una relación en tercera forma normal, es el presentado por Phillip A. Bernstein. Según S. Ceri y G. Gottlob[D]. El mejor algoritmo de normalización del que se tiene conocimiento es el de Bernstein[B], ya que éste demuestra que además de obtener relaciones en tercera forma normal, el número de relaciones resultante es el menor posible.

La mayoría de los algoritmos de normalización, construyen relaciones en tercera forma normal directamente sin descomposiciones intermedias en segunda forma normal. esto se debe a que dadas las características del algoritmo de normalización, se tiene toda la información necesaria para descomponer directamente en tercera forma normal. Si se realiza una descomposición intermedia en segunda forma normal, se tendrá que aplicar el algoritmo de descomposición en tercera forma normal a cada una de las nuevas relaciones generadas por la descomposición en segunda forma normal, lo cual no es eficiente.

El algoritmo de Bernstein consiste de los siguientes pasos:

1. Encontrar la cobertura no redundante (H) del conjunto inicial de dependencias funcionales.

2. Dividir el conjunto de dependencias H en grupos de Hi de tal forma que todas las dependencias en cada grupo tengan partes determinantes iguales.

3. Mezclar grupos Hi y Hj con partes determinantes X y Y respectivamente, cuando las dependencias X ⟶ Y y Y ⟶ X sean válidas. X y Y son llaves equivalentes.

4. Construir una cobertura particular de H que incluya todas las dependencias X⟶Y, donde Y y X son llaves equivalentes y todas las demás dependencias no sean redundantes.

5. Construir relaciones.

El paso 1 se obtiene con el proceso de cobertura no redundante expuesto anteriormente. Al observar que nuestras dependencias funcionales

contienen sólo un atributo como parte determinada, la eliminación de atributos extraños es comparable a la eliminación de dependencias funcionales parciales que se maneja generalmente en el contexto de normalización, por ejemplo:

Sea la eliminación del atributo extraño:

$$A, B \longrightarrow D$$

y

$$(A, B) - (A) \longrightarrow D$$

esto es

$$B \longrightarrow D$$

significando que D depende parcialmente de A y B.

El paso 2 realiza la agrupación de dependencias funcionales con partes determinantes iguales, por ejemplo:

Sean

$$A, B \longrightarrow C$$

$$A, B \longrightarrow D$$

$$E, F \longrightarrow G$$

$$E, F \longrightarrow H$$

$$I \longrightarrow J$$

obteniendo tres grupos representativos

grupo 1 : (A, B)

grupo 2 : (E, F)

grupo 3 : (I)

El paso 3 encuentra llaves equivalentes y forma un sólo grupo de ambas llaves, por ejemplo:

Sean: A,B —· C, D, E, F

C,D —· A, B, H, I

entonces

A, B C, D son llaves equivalentes

y de los grupos existentes se forma uno solo

grupo (A, B)

grupo ((A, B), (C, D))

grupo (C, D)

El paso 4 elimina las dependencias transitivas y agrega las dependencias resultantes a la obtención de las llaves equivalentes, por ejemplo.

(A, B) (C,D).

Para eliminar las dependencias transitivas se utiliza un proceso similar al de eliminación de dependencias redundantes, por ejemplo:

Si tenemos A, B —· C, D, E

y D —· E

eliminamos entonces la dependencia funcional temporalmente

A. B —· E

Sin embargo observamos que E se puede conocer a partir de:

D —· E

con lo que eliminamos definitivamente la dependencia transitiva:

A, B —· E

quedando como resultado:

$$A, B \longrightarrow C, D$$

y

$$D \longrightarrow E$$

El paso 5 construye nuevas relaciones a partir de cada grupo. Este paso tiene que ver con la creación de nuevos nombres para las nuevas relaciones, genera los esquemas de las nuevas relaciones y obtiene algunas llaves de cada una de ellas.

Las llaves están dadas por la parte determinante que caracteriza a cada grupo y a través de las dependencias funcionales de la base de conocimiento se recopilan todos los atributos que pertenecen a cada nuevo esquema de las relaciones generadas.

Una vez generadas las nuevas relaciones en tercera forma normal, éstas son cargadas a la base de conocimiento del paquete junto con las llaves y la información acerca de la descomposición.

1.5 Codificación.

A continuación se muestra el código fuente del paquete normalizador, el cual está dividido en dos módulos. El primero es el módulo que se encarga de realizar el proceso de normalización y el segundo realiza todo el manejo de pantallas referentes a la interface con el usuario.

```
/*****************************************************************/
/* TESIS.                                                        */
/*                                                               */
/* Sistema : Paquete Normalizador.                               */
/*                                                               */
/* Módulo  : Normalización.                                      */
/*                                                               */
/*                                                               */
/* Autores : Cárdenas, De Uriarte, Rosales.                      */
/*                                                               */
/* Fecha   : Enero/1989                                          */
/*                                                               */
/* Descripción General:                                          */
/*                                                               */
/*                                                               */
/* Este programa recibe como entrada el esquema de una           */
/* relación : esquema(nombre_rel,[atrib1,atrib2,..,atribn])      */
/* y las dependencias funcionales entre atributos :              */
/* depfun(nombre_rel,[atirb1,..atribi],[atrib3,..,atribk])       */
/* donde:                                                        */
/*                                                               */
/* nombre_rel es el nombre de la relación a normalizar.          */
/*                                                               */
/* En las dependencias funcionales la primera lista de a-        */
/* tributos es la parte determinante y la segunda, la            */
/* parte determinada.                                            */
/*                                                               */
/* Los datos de entrada se obtienen por medio del módulo         */
/* de interface y se cargan a la base de datos antes de ini-     */
/* ciar programa.                                                */
/*                                                               */
/* Con los datos iniciales, se normaliza la relación y se        */
/* descompone en una o más relaciones en tercera forma nor -     */
/* mal y se obtienen algunas llaves de dichas relaciones.        */
/*                                                               */
/*****************************************************************/

/* Directivas del Compilador */

code = 3072
nowarnings

/* Identificación del módulo */

project "TESIS"
include "globndef.pro"
include "nhelp.pro"

PREDICATES
```

```
elemento(STRING,STRINGLIST)
elemento(STRINGLIST,listas)
subconjunto(STRINGLIST,STRINGLIST)
atributo(STRINGLIST,STRING)
unions(STRINGLIST,STRINGLIST,STRINGLIST,STRINGLIST)
union(STRINGLIST,STRINGLIST,STRINGLIST,STRING).
menos1(STRINGLIST,STRINGLIST,STRING)
menos1(listas,listas,STRINGLIST)
menos(STRINGLIST,STRINGLIST,STRINGLIST)
closure(STRING,STRINGLIST,STRINGLIST)
elim_atrib(STRING)
reducelhs(STRING,STRINGLIST,STRINGLIST,STRINGLIST)
elim_dep_fun_red(STRING)
encuentra_cov_min(STRING)
thirdnf(STRING)
paso1(STRING)
paso2(STRING)
paso3(STRING)
paso4(STRING)
paso5_a(STRING)
paso5(STRING,integer)
remembercovering(STRING)
mezcla(STRING,STRINGLIST,STRINGLIST)
yaexistegrupo(STRING,STRINGLIST,STRINGLIST)
elimin(STRING,STRINGLIST)
encuentranombre(STRING,integer,STRING)
guardaalgunallave(STRING,listas)
hazesquema(STRING,STRING,listas)
collect1(STRING,listas,STRINGLIST)
collect(STRING,listas,STRINGLIST,STRINGLIST,STRINGLIST)
quitamodfds(STRING)
reassertrememberedfds.
esatribvalid(STRING,listas,STRING)
aux1(STRING,STRINGLIST,STRINGLIST,STRINGLIST)
aux2(STRING,STRINGLIST,STRINGLIST,STRINGLIST)
borra(STRING,STRINGLIST,STRINGLIST)
aux3(STRING,listas,STRING,STRINGLIST,STRINGLIST)
aux4(STRINGLIST,STRINGLIST,STRING)
recorre(integer)
inicianorm(STRING)
imprime(string)
inicial(string)
junta(string)
desp_msg(integer,integer)
esc_gpo(integer,integer,listas)
mensaje(integer,integer,string,stringlist,stringlist,stringlist,
    listas)
enciende
restaura

CLAUSES

normaliza:- inicianorm(REL),
        not(REL = ""),!,
        paso1(REL),
        thirdnf(REL),
        restaura,
        recorre(4),
        removewindow,
        recorre(5),
        removewindow,
        recorre(6),
        removewindow,
        recorre(2).

normaliza.
 /* inicializacion de ventanas */
```

```
restaura:-retract(tipo_norm(_)),fail.
restaura.

inicianorm(REL):-esquema(REL,_),!,
      concat(" DESCOMPOSICION DE ",REL,F),
      concat(F," EN 3A FORMA NORMAL ",TITULO),
      makewindow(2,31,113,TITULO,3,0,21,80),
      makewindow(1,30,0,"",4,1,3,78),
      makewindow(5,56,0,"",7,1,12,78),
      makewindow(6,31,0,"",19,1,4,78),
      changestatus(TITULO),
      consult("mensajes.crt").

inicianorm(REL):-not(esquema(_,_)),!,REL = "".

Imp_dep(LHS,RHS,REN):-concatena(["( "],LHS,ILHS),
              concatena(ILHS,[") - ("],NL),
              concatena(NL,RHS,DEP),
              concatena(DEP,[" )"],DEPFUN),
              escribe_lista(DEPFUN,1,76,REN).

escribe_lista([],Origen,_,REN):-!,NREN = REN + 1,
                  cursor(NREN,Origen).

escribe_lista([Cabeza|Resto],Origen,Limite,REN):-
                  str_len(Cabeza,LL),
                    cursor(_,COL),
                (COL + LL + 1) Limite,!,
                    write(Cabeza," "),
              escribe_lista(Resto,Origen,Limite,REN).

escribe_lista(Lista,Origen,Limite,REN):-RREN = REN + 1,
                    cursor(RREN,Origen),
            escribe_lista(Lista,Origen,Limite,RREN).


recorre(VN):-gotowindow(VN),clearwindow.


/* Funciones de uso común : utileria */

 /* elemento(E,L) : E es elemento de L */

     elemento(E,[E|T]).
     elemento(E,[_|T]):- elemento(E,T).

 /* subconjunto(A,B) : A esta contenido en B */

     subconjunto([],X):-!.
     subconjunto([H|TA],[H|TB]):-!,subconjunto(TA,TB).
     subconjunto(A,[_|TB]):-subconjunto(A,TB).

 /* atributo(E,R) : E es una lista de atributos de R */

     atributo(E,R):- esquema(R,S), subconjunto(E,S)

 /* unions(U,A,B,L) : Unión de dos subconjuntos A y B
   que pertenecen a L, dejando el resultado en U */

     unions(A,A,[],L):-!.
     unions(B,[],B,L):-!.
     unions([HL|TU],[HL|TA],[HL|TB],[HL|TL]):-!,
                  unions(TU,TA,TB,TL).
     unions([HL|TU],[HL|TA],B,[HL|TL]):-!,
                  unions(TU,TA,B,TL).
     unions([HL|TU],A,[HL|TB],[HL|TL]):-!,
```

```
                    unions(TU,A,TB,TL).
  unions(U,A,B,[HL|TL]):-unions(U,A,B,TL).

/* union(U,A,B,REL) : Unión de dos conjuntos de
   atributos ordenados, A y B, que pertenecen a REL
   obteniendo el resultado en U */

  union(U,A,B,REL):-esquema(REL,L),
                    subconjunto(A,L),
                    subconjunto(B,L),
                    unions(U,A,B,L).

/* Menos1(R,A,B) : substrae el elemento B del conjunto
   A obteniendo el resultado en R */

  menos1([],[],B):-!.
  menos1([A],[B|TA],B):-!.
  menos1([HA|TX],[HA|TA],B):-menos1(TX,TA,B).

/* Menos(R,A,B) : R = A - B , donde A y B son conjuntos
   ordenados */

  menos(R,R,[]):-!.
  menos(Z,A,[HB|TB]):-menos1(R,A,HB),menos(Z,R,TB).

/* concatena(X,Y,Z) : Concatenación de X y Y obteniendo
   el resultado en Z */

  concatena([],L,L).
  concatena([X|L1],L2,[X|L3]):-concatena(L1,L2,L3).

/* Obtención del Closure de un conjunto de atributos */

  closure(REL,X,RESULTADO):-depfun(REL,LHS,RHS),
                    subconjunto(LHS,X),
                    not (subconjunto(RHS,X)),
                    union(W,X,RHS,REL),!,
                    closure(REL,W,RESULTADO).


  closure(REL,X,RESULTADO):-RESULTADO = X.


/* Descripción de las reglas de Normalización */


  /* Cobertura no redundante */

  /* a) Eliminación de atributos extraños */

  elim_atrib(REL):-mensaje(4,41,"",[],[],[],[]),
                   depfun(REL,LHS,RHS),
                   reducelhs(REL,LHS,RHS,NUEVALHS),
                   not (LHS = NUEVALHS),
                   retract(depfun(REL,LHS,RHS)),
                   asserta(depfun(REL,NUEVALHS,RHS)),
                   mensaje(6,62,"",NUEVALHS,[],[],[]),
                   fail.

  elim_atrib(REL):-mensaje(4,42,"",[],[],[],[]).



  continua:-changestatus(
          "        OPRIMA CUALQUIER TECLA PARA CONTINUAR"),
        shiftwindow(A),
        gotowindow(2),
```

```
        cursor(18,77),sound(5,10000),
        readchar(_),
        shiftwindow(A),
        changestatus(
*                                   *).

reducelhs(REL,LHS,RHS,NUEVALHS):- elemento(A,LHS),
                    menos1(Z,LHS,A),
                    not (Z = []),
                    closure(REL,Z,ZCLO),
                mensaje(5,51,A,LHS,RHS,ZCLO,[]),
                    subconjunto(RHS,ZCLO),
                    !,
                    elemento(RR,RHS),
                mensaje(6,61,RR,[],[],[],[]),
                reducelhs(REL,Z,RHS,NUEVALHS).

reducelhs(REL,LHS,RHS,NUEVALHS):-NUEVALHS = LHS.


/* b) Eliminación de dependencias funcionales
    redundantes */

elim_dep_fun_red(REL):-mensaje(4,43,"",[],[],[],[]),
                depfun(REL,LHS,RHS),
                borra(REL,LHS,RHS),
                closure(REL,LHS,Z),
                mensaje(5,52,"",LHS,RHS,Z,[]),
                aux1(REL,LHS,RHS,Z),
                fail.

elim_dep_fun_red(REL):-mensaje(4,44,"",[],[],[],[]).

aux1(REL,LHS,RHS,Z):-not(subconjunto(RHS,Z)),!,
                elemento(RR,RHS),
                mensaje(6,63,RR,[],[],[],[]),
                asserta(depfun(REL,LHS,RHS)).

aux1(REL,LHS,RHS,Z):-!,subconjunto(RHS,Z),
                elemento(RR,RHS),
                mensaje(6,64,RR,[],[],[],[]).



/* Combinando a) y b) para encontrar la cobertura
  mínima */

encuentra_cob_min(REL):-elim_atrib(REL),
                elim_dep_fun_red(REL).


/* algoritmo de Berenstein para descomponer una
  relación en 3a forma normal */

thirdnf(REL):- remembercovering(REL),
          mensaje(4,45,"",[],[],[],[]),
          paso2(REL),
          mensaje(4,47,"",[],[],[],[]),
          paso3(REL),
          mensaje(4,49,"",[],[],[],[]),
          paso4(REL),
          /*enciende, */
          mensaje(4,411,"",[],[],[],[]),
          paso5_a(REL).

enciende:-retract(tipo_norm(_)),fail.
```

```
enciende:-l,asserta(tipo_norm('T')).

/* paso 1   encontrar una cobertura no redundante */

paso1(REL):-encuentra_cob_min(REL),
        remembercovering(REL)


remembercovering(REL):-depfun(REL,LHS,RHS),
            assertz(remember(depfun(REL,LHS,RHS))),
                fail

remembercovering(REL).

/* paso 2 : división de la cobertura en grupos con
   determinantes iguales */

paso2(REL):-depfun(REL,LHS,_).
        not (grupo(REL,[LHS])).
        asserta(grupo(REL,[LHS]))
        closure(REL,LHS,CLO).
        mensaje(5,53,"",LHS,[],CLO,[]).
        asserta(clo(REL,LHS,CLO))
        fail.

paso2(REL):-mensaje(4,46,"",[],[],[],[]).

/* paso 3 : mezcla de grupos con llaves equivalentes;
   depfunj son las dependencias de paso del conjunto
   J mencionado en la referencia de Berenstain */

paso3(REL):-clo(REL,LHS1,LHS1CLO),
        clo(REL,LHS2,LHS2CLO).
        not (LHS1 = LHS2),
        subconjunto(LHS2,LHS1CLO),
        subconjunto(LHS1,LHS2CLO),
        not(yaexistegrupo(REL,LHS1,LHS2)),
        mensaje(6,65,"",[],[],[],[]),
        mensaje(5,54,"",LHS1,LHS2,[],[]),
        mezcla(REL,LHS1,LHS2),
        asserta(depfunj(REL,LHS1,LHS2)),
        asserta(depfunj(REL,LHS2,LHS1)).
        fail.

paso3(REL):-clo(REL,LHS,LHSCLO).
        retract(clo(REL,LHS,LHSCLO)),
        fail.

paso3(REL):-depfunj(REL,LHS,RHS)
        depfun(REL,LHS,A),
        subconjunto(A,RHS),
        retract(depfun(REL,LHS,A)).
        fail.

paso3(REL):-mensaje(4,48,"",[],[],[],[])


mezcla(REL,LHS1,LHS2):- grupo(REL,G1),
                elemento(LHS1,G1),
                grupo(REL,G2),
                elemento(LHS2,G2),
                retract(grupo(REL,G1)),
                retract(grupo(REL,G2)),
                concatena(G1,G2,NVOGPO),
                asserta(grupo(REL,NVOGPO)),
                mensaje(5,55,"",[],[],[],NVOGPO).
```

```
yaexistegrupo(REL,LHS1,LHS2):- grupo(REL,GRUPO),
                    elemento(LHS1,GRUPO),
                    elemento(LHS2,GRUPO).

/* paso 4 : eliminación de dependencias transitivas */

paso4(REL):-depfunj(REL,LHS,RHS),
        assertz(depfun(REL,LHS,RHS)),
        fail.


/* prueba de redundancia para las dependencias que no
   estan en J */

paso4(REL):-depfun(REL,LHS,RHS),
        not(depfunj(REL,LHS,RHS)),
        borra(REL,LHS,RHS),
        closure(REL,LHS,Z),
        mensaje(5,52,"",LHS,RHS,Z,[]),
        aux2(REL,LHS,RHS,Z),
        fail.

paso4(REL):-depfunj(REL,LHS,RHS),
        retract(depfun(REL,LHS,RHS)),
        fail.


paso4(REL):-tipo_norm(TN),TN = 'T',
        mensaje(5,56,"",[],[],[],[]),
        grupo(REL,G),
        write("GRUPO :"),
        cursor(A,B),
        esc_gpo(A,B,G),
        cursor(AA,BB),NB = BB - 7,
        cursor(AA,NB),
        fail.

paso4(REL):-!,mensaje(4,410,"",[],[],[],[]).

borra(REL,LHS,RHS):-retract(depfun(REL,LHS,RHS)),!.

aux2(REL,LHS,RHS,Z):-not(subconjunto(RHS,Z)),!,
            elemento(RR,RHS),
            mensaje(6,63,RR,[],[],[],[]),
          asserta(depfun(REL,LHS,RHS)).

aux2(REL,LHS,RHS,Z):-!,subconjunto(RHS,Z),
            elimin(REL,LHS),
            elemento(RR,RHS),
            mensaje(6,64,RR,[],[],[],[]).



/* eliminación de un elemento LHS de un grupo G */

elimin(REL,LHS):-not(depfun(REL,LHS,_)),
            grupo(REL,G),
            elemento(LHS,G),
            retract(grupo(REL,G)),
            menos1(Z,G,LHS),
            not(Z = []),
            asserta(grupo(REL,Z)),!.

elimin(REL,LHS).

/* paso 5 : transformación de cada grupo en una
   relación en 3a forma normal */
```

```
paso5_a(REL):-paso5(REL,0),
        enciende,
/*        mensaje(4,411,"",[],[],[],[]),*/
        junta(REL),
        inicial(REL),
        imprime(REL),
        mensaje(4,412,"",[],[],[],[]).

paso5(REL,NR):-grupo(REL,G),
        NVONR = NR + 1,
        encuentranombre(REL,NVONR,NVAREL),
        hazesquema(REL,NVAREL,G),
        guardaalgunallave(NVAREL,G),
        assertz(decomp(REL,NVAREL)),
        assertz(en3afn(NVAREL)),
        assertz(tnfdecomp(REL,NVAREL)),
        retract(grupo(REL,G)),!,
        paso5(REL,NVONR).

paso5(REL,NR):-quitamodfds(REL),
        reassertrememberedfds.

encuentranombre(REL,NR,NVAREL):-concat(REL,"_",PASO),
                SUFIJO = NR+96,
                char_int(CHAR,SUFIJO),
                str_char(SCHAR,CHAR),
                concat(PASO,SCHAR,NOMBRE),
                NVAREL = NOMBRE.

guardaalgunallave(NVAREL,G):-elemento(K,G),
                assertz(llave(NVAREL,K)),
                fail.

guardaalgunallave(NVAREL,G).

hazesquema(REL,NVAREL,G):- collect1(REL,G,NVOESQUEMA),
        assertz(esquema(NVAREL,NVOESQUEMA)).

/* colectando en RESULTADO el esquema de la relación
  sintetizada asociada al grupo G.  Un atributo A
  pertenece al esquema si y sólo si pertenece al
  determinante, o determinado de alguna  dependencia
  funcional cuyo determinante está en G */

collect1(REL,G,RESULTADO):-esquema(REL,TOTEST),
        collect(REL,G,TOTEST,[],RESULTADO).

collect(REL,G,TOTEST,ACCEPT,RES):-elemento(A,TOTEST),
        menos1(NEWTOTEST,TOTEST,A),
        aux3(REL,G,A,ACCEPT,NEWACCEPT),
        collect(REL,G,NEWTOTEST,NEWACCEPT,RES).

collect(REL,G,TOTEST,ACCEPT,RES):-RES = ACCEPT.

esatribvalid(REL,G,A):-!,elemento(LHS,G),
        depfun(REL,LHS,RHS),
        aux4(LHS,RHS,A).


aux3(REL,G,A,ACCEPT,NEWACCEPT):-esatribvalid(REL,G,A),
                !,
        union(NEWACCEPT,ACCEPT,[A],REL).

aux3(REL,G,A,ACCEPT,NEWACCEPT):-!,NEWACCEPT = ACCEPT.

aux4(LHS,RHS,A):-elemento(A,LHS),!.
```

```
    aux4(LHS,RHS,A):-!,elemento(A,RHS).

    quitamodfds(REL):-depfun(REL,LHS,RHS),
            retract(depfun(REL,LHS,RHS)),fail.

    quitamodfds(REL):-depfunj(REL,LHS,RHS),
            retract(depfunj(REL,LHS,RHS)),fail.

    quitamodfds(REL).

    reassertrememberedfds:-remember(depfun(REL,LHS,RHS)),
                assertz(depfun(REL,LHS,RHS)),
            retract(remember(depfun(REL,LHS,RHS)))
                ,fail.

    reassertrememberedfds.

inicial(RELO):-esquema(NR,ATR),
        RELO = NR,!,
        mensaje(5,58,NR,ATR,[],[],[]),
        field_str(0,30,16,"ESQUEMA INICIAL"),
        field_attr(0,30,16,48),
        continua.

imprime(RELO):-esquema(NOMBRE,ATRIBS),
        not(NOMBRE = RELO),
        decomp(RELO,NOMBRE),
        mensaje(5,58,NOMBRE,ATRIBS,[],[],[]),
        imp_llaves(NOMBRE),
        fail.
imprime(_).

imp_llaves(REL):-llave(REL,LLAVE),
          cursor(A,B),
          concatena(["Llave : "],LLAVE,LL),
          esc_lis(LL,A,B),
          fail.

imp_llaves(REL):-continua.


/* Predicados auxiliares para desplegar información */

desp_msg(NV,NMENS):-tipo_norm(A),m(NMENS,MSG),A = 'T',
          !,recorre(NV),
          cursor(0,0),
          window_str(MSG).

esc_atrib(ATRIB,REN,COL,CRTAT):-!,str_len(ATRIB,LEN),
                field_attr(REN,COL,LEN,CRTAT),
                field_str(REN,COL,LEN,ATRIB).

esc_lis(LISTA,REN,COL):-!,cursor(REN,COL),
            concatena(["("],LISTA,L1),
            concatena(L1,[")"],L2),
            escribe_lista(L2,COL,76,REN).

junta(REL):-mensaje(5,57,REL,[],[],[],[]),
      decomp(REL,NREL),
      cursor(A,B),
      esc_atrib(NREL,A,B,56),
      AA = A + 1,
      cursor(AA,B),
      fail.

junta(REL):-continua.
```

```
esc_gpo(_,_,[]):-!.
esc_gpo(R,C,[Cabeza|Resto]):-tipo_norm(NA),NA = 'T',!,
                    esc_lis(Cabeza,R,C),
                    cursor(A,B),
                    esc_gpo(A,B,Resto).

mensaje(4,410,"",[],[],[],[]):-desp_msg(4,410),!,
                     recorre(6),
                     gotowindow(4),
                     field_attr(2,2,2,117),
                     continua.

mensaje(4,NM,"",[],[],[],[]):-desp_msg(4,NM),!,
                    recorre(5),
                    recorre(6),
                    gotowindow(4),
                    field_attr(2,2,2,117),
                    continua.

mensaje(5,51,ATR,LHS,RHS,ZCLO,[]):-desp_msg(5,51),!,
                     field_attr(1,1,1,62),
                     field_attr(6,1,1,62).
                     field_attr(7,1,1,62)
                     cursor(2,1).
                     imp_dep(LHS,RHS,2),
                     esc_atrib(ATR,6,37,62),
                     esc_lis(ZCLO,8,1),
                     continua.

mensaje(5,52,"",LHS,RHS,ZCLO,[]):-desp_msg(5,52),!,
                     field_attr(1,1,1,62).
                     field_attr(6,1,1,62),
                     cursor(2,1),
                     imp_dep(LHS,RHS,2),
                     esc_lis(ZCLO,7,1)

mensaje(5,53,"",LHS,[],ZCLO,[]):-desp_msg(5,53),!,
                    field_attr(1,1,1,62),
                    field_attr(6,1,1,62).
                    esc_lis(LHS,2,1),
                    esc_lis(ZCLO,7,1),
                    continua.

mensaje(5,54,"",LHS1,LHS2,[],[]):-desp_msg(5,54),!,
                     field_attr(1,1,1,62),
                     cursor(2,1),
                     imp_dep(LHS1,LHS2,2),
                     cursor(A,_),
                     imp_dep(LHS2,LHS1,A)
                     continua.

mensaje(5,55,"",[],[],[],NVOGPO):-desp_msg(5,55),!,
                    field_attr(1,1,1,62),
                    cursor(2,1),
                    esc_gpo(2,1,NVOGPO),
                    continua,
                    recorre(6).

mensaje(5,56,"",[],[],[],[]):-desp_msg(5,56),!
                   field_attr(1,1,1,62),
                   cursor(2,1).

mensaje(5,57,REL,[],[],[],[]):-desp_msg(5,57),!
                    field_attr(1,1,1,62),
                    esc_atrib(REL,1,17,62),
                    cursor(3,15)
```

```
mensaje(5,58,REL,ATRIBS,[],[],[]):-desp_msg(5,58),!,
                    esc_atrib(REL,2,45,62),
                    esc_atrib(REL,4,5,56),
                    str_len(REL,L),
                    C = 5 + L + 1,
                    cursor(4,C),
                    esc_lis(ATRIBS,4,C).

mensaje(6,61,ATR,[],[],[],[]):-desp_msg(6,61),!,
                  field_attr(0,0,78,116),
                  esc_atrib(ATR,1,15,126),
                  continua.

mensaje(6,62,"",NLHS,[],[],[]):-desp_msg(6,62),!,
                   field_attr(0,1,1,116),
                   cursor(1,1),
                   esc_lis(NLHS,1,1),
                   continua,
                   recorre(6).

mensaje(6,63,ATR,[],[],[],[]):-desp_msg(6,63),!,
                  field_attr(0,1,1,116),
                  esc_atrib(ATR,0,44,126),
                  continua.

mensaje(6,64,ATR,[],[],[],[]):-desp_msg(6,64),!,
                  field_attr(0,0,78,116),
                  esc_atrib(ATR,1,10,126),
                  continua.

mensaje(6,65,"",[],[],[],[]):-desp_msg(6,65),!,
                 field_attr(0,1,1,116),
                 field_attr(0,36,1,116).

mensaje(_,_,_,_,_,_,_):-!.

/* Fin Módulo Normalización */
/*****************************************************************/
```

```
/*****************************************************************/
/* TESIS.                                                        */
/*                                                               */
/* Sistema : Paquete Normalizador                                */
/*                                                               */
/* Módulo  : Interface de Usuario                                */
/*                                                               */
/* Autores : Cárdenas, De Uriarte, Rosales.                      */
/*                                                               */
/* Fecha   . Enero/1989.                                         */
/*                                                               */
/* Descripción General:                                          */
/*                                                               */
/*   Esta interface permite al usuario interactuar con el Nor-   */
/* malizador para poder dar de alta datos, normalizar, observar  */
/* resultados, imprimir, etc. de una manera amigable.            */
/*                                                               */
/*   El programa se basa principalmente en un menú PULLDOWN      */
/* donde se tienen 4 opciones generales:                         */
/*                                                               */
/* Edición.                                                      */
/*                                                               */
/* En esta opción se tiene el siguiente submenú.                 */
/*                                                               */
/*   > Edición de Esquema: Para dar de alta un esquema nuevo.    */
/*   > Edición de Dependencias: Para dar de alta/baja depen -    */
/*     dencias funcionales asociadas a un esquema dado.          */
/*                                                               */
/* Base de Datos.                                                */
/*                                                               */
/* En esta opción se tienen todas las operaciones para manejar   */
/* los datos de la base, controladas por el siguiente submenú:   */
/*                                                               */
/*   > Carga de datos a la Base: Para cargar esquema y depen-    */
/*     dencias.                                                  */
/*   > Almacenar en disco : Para almacenar todos los datos de    */
/*     la base en un archivo en disco.                           */
/*   > Despliega datos: Para desplegar todos los datos carga-    */
/*     dos en la base.                                           */
/*   > Imprime : Para imprimir los datos de la base.             */
/*   > Borra : Para eliminar todos los datos de la base.         */
/*                                                               */
/* Normalización.                                                */
/*                                                               */
/* Esta opción tiene 2 modalidades : Tutorial y Directa .        */
/*                                                               */
/* Terminar.                                                     */
/* Opción de salida del Normalizador                             */
/*                                                               */
/*****************************************************************/


/* Directivas del Compilador */

nowarnings
code = 8000

/* Identificación del  Módulo */

project "TESIS"
include "globndef.pro"     /* Dominios y predicados globales */

/* Inclusiones de utilerías del PROLOG Toolbox y archivos
   auxiliares */

include "nhelp.pro"    /* Utileria para manejo de ayudas */
include "pulldown.pro" /* Utileria para manejo del Pulldown */
```

```
include "nmenu.pro"    /* Utilerias para manejo de menues */
include "nboxmenu.pro"
include "status.pro"
include "nlineinp.pro" /* Utileria de capatura */

PREDICATES

/* Lectura Esquema */

leeEsquema
lee_atributos(STRINGLIST,STRINGLIST)
verifica(STRING,STRING,STRING,LEN,ROW,COL)
almacena(STRING,STRINGLIST)
cambia(ROW,COL)

/* Lectura de Dependencias */

leeDependencias
inicia
leeDep(STRINGLIST,STRINGLIST,STRINGLIST)
deps(STRING,STRINGLIST)
checa_1_atributo(STRING,STRINGLIST,STRINGLIST)
recdeps(INTEGERLIST,STRINGLIST,STRINGLIST,STRINGLIST)
invierte(INTEGERLIST,INTEGERLIST,INTEGERLIST)
obtiene(STRINGLIST,integer,integer,STRING)


/* Desplegado de Información */

imprime_esquema
imp_dependencias
fds
verif(integer,integer)
despliega
despliega_esquema(STRING)
despliega_deps(STRING)
despliega_decomp(STRING)

/* Control */

lee_archivos(STRING,STRING,STRING)
guarda
checa_long(STRING,STRING)
almacena_archivo(integer,STRING)
wesq
wdeps
wllave
wdecomp
wen3afn
wtnfd
edita(integer,STRING,STRING)
versino(char,STRING,STRINGLIST,STRINGLIST)
opcion (integer)
no_existe(STRING,STRING)
borra_datos
borraesq
borradep
borragpo
borrallave
borradecomp
borraen3afn
borratnfd
carga(INTEGER)
carga1(INTEGER,STRING)
quita_vent
checa(STRING)
```

```
/* Impresión */

impresion
imp_esq_orig(STRING)
imp_esq(STRING,STRINGLIST)
imp_depfun(STRING)
imp_decomp(STRING)
llaves(STRING)
ifds
espacios(integer)
imp_lista(STRINGLIST,integer,integer,integer)

GOAL

principal

CLAUSES

/* META PRINCIPAL */

/* Inicialización de ventanas, estatus y Pulldown menú */

principal:-assertz(helpfile("ayudas.hlp")),
        consult("ayudas.def"),
        makewindow(1,31,113,"",3,0,21,80),
        makestatus(31,
"F1 = AYUDA  SELECCION = \24 \25 \26 ,27 y \17\217   ó bien con la primera letra"),
        push_helpcontext(pull),
        pulldown(113,
        [curtain(7,"Edición",["Esquema","Dependencias"]),
         curtain(28,"Base de Datos",["Carga"
                  "Almacena en disco"
                  "Despliega",
                  "Borra",
                  "Imprime"]),
         curtain(52,"Normalización",["Tutorial","Directa"]),
         curtain(69,"Terminar",[])],El.EC,SUB.ELEC),
         pop_helpcontext,exit

/* Acción a seguir según opción del Pulldown menú */

/* Creación de un nuevo Esquema */

pdwaction(1,1) -makewindow(2,31,113
          " Creación de Nuevo Esquema
          3,0,21,80),
          clearwindow,
          borra_datos,
          leeEsquema
          imprime_esquema
          clearwindow,
          leeDependencias
          imp_dependencias
          guarda,
          changestatus(
"F1 = AYUDA  SELECCION = \24 \25 \26 \27 y  \17\217   ó bien con la primera letra"),
          removewindow,

pdwaction(1,1):-removewindow
          changestatus(
"F1 = AYUDA  SELECCION = \24 \25 \26 \27 y  \17\217   ó bien con la primera letra")

/* Edición de Dependencias */

pdwaction(1,2):-borra_datos,
          push_helpcontext(menuedit),
          menu(5,15,116,113,["Alta","Baja"],"",1,OP),
```

```
        pop_helpcontext,
        not(OP = 0),
        makewindow(2,31,113,
        " Edición de Dependencias Funcionales ",
        3,0,21,80),
        clearwindow,
        opcion(OP),
        changestatus(
"F1 = AYUDA  SELECCION = \24 \25 \26 \27 y  \17\217  ó bien con la primera letra"),
        removewindow.

pdwaction(1,2):-!.

/* Carga de Datos a la base */

pdwaction(2,1):-makewindow(2,31,113,
        " Carga de Datos a la Base ",3,0,21,80),
        push_helpcontext(menucarga),
        menu(4,35,116,113,
   ["Esquema","Dependencias","DatosNormalización"],"",1,OP),
        pop_helpcontext,
        not(OP = 0),
        clearwindow,
        carga(OP),
        changestatus(
"F1 = AYUDA  SELECCION = \24 \25 \26 \27 y  \17\217  ó bien con la primera letra"),
        removewindow.

pdwaction(2,1):-!,removewindow.

/* Almacena Datos de la base en Disco */

pdwaction(2,2):-lee_archivos("",NBD,"BDN"),
        not(NBD = ""),!,
        almacena_archivo(3,NBD),
        changestatus(
"F1 = AYUDA  SELECCION = \24 \25 \26 \27 y  \17\217  ó bien con la primera letra").

pdwaction(2,2):-!.

/* Despliega datos de la base */

pdwaction(2,3):-shiftwindow(A),despliega,
        changestatus(
"F1 = AYUDA  SELECCION = \24 \25 \26 \27 y  \17\217  ó bien con la primera letra"),
        shiftwindow(A).
pdwaction(2,3):-!.

/* Limpia los datos de la base */

pdwaction(2,4):-borra_datos.

/* Imprime los datos de la Base */

pdwaction(2,5):-shiftwindow(A),
        push_helpcontext(impresora),
        help,
        writedevice(OLD),
        writedevice(printer),
        impresion,
        writedevice(OLD),
        shiftwindow(A).
pdwaction(2,5).

/* Normalizacion Tutorial */

pdwaction(3,1):-!,shiftwindow(A),
```

```
                asserta(tipo_norm('T')),
                normaliza,
                refreshstatus,
                removewindow,
                shiftwindow(A).
pdwaction(3,1).

/* Normalizacion Directa */

pdwaction(3,2):-!,shiftwindow(A),
                asserta(tipo_norm('D')),
                normaliza,
                removewindow,
                shiftwindow(A),
                refreshstatus.

pdwaction(3,2).

/* Salida */

pdwaction(4,0):-!,shiftwindow(A),
                gotowindow(1),
                removestatus,
                removewindow,
                shiftwindow(A),
                removewindow,
                exit.


/* Lectura Esquema Inicial */

leeEsquema:-changestatus(
"F1 = AYUDA  ESC = CANCELA          Digite Esquema Inicial por normalizar"),
        push_helpcontext(esquema),
        field_str(1,27,23,"NOMBRE DE LA RELACION :"),
        lineinput_leave(7,29,20,116,0,"","",NR),
        verifica(NR,REL,"",20,7,29),
        gotowindow(2),
        field_str(5,33,12,"ATRIBUTOS :"),
        cursor(12,6),
        changestatus(
"F1 = AYUDA  ESC = TERMINAR   Digite los Atributos de la Relación y \17\217"),
        lee_atributos(ATRIB,[]),
        not(ATRIB = []),!,
        almacena(REL,ATRIB),
        pop_helpcontext,
        clearwindow.

leeEsquema.


/* Verificación de nombre válido en PROLOG */

verifica(NOM,REL,_,LEN,_,_):-isname(NOM),
                str_len(NOM,LONG),
                LONG 0,
                LONG LEN,!,
                REL = NOM.

verifica(NOM,REL,_,LEN,_,_):-isname(NOM),
                str_len(NOM,LONG),
                LONG LEN,!,
                frontstr(LEN,NOM,REL,_).

verifica(NOM,REL,PROM,LEN,REN,COL):-!,
        changestatus(
"F1 = AYUDA      ** NOMBRE INVALIDO **      DIGITE NUEVAMENTE"),
```

```
            sound(5,200),
            help,
            lineinput_leave(REN,COL,LEN,116,0,PROM,NOM,N),
            verifica(N,REL,PROM,LEN,REN,COL).

/* Lee Atributos de la relación */

lee_atributos(ATR,ANTERIOR):- cursor(A,B),
                    cambia(A,B),
                    cursor(AA,BB),
                    not(BB = 16),
                    not(AA = 54),
                    lineinput_leave(AA,BB,15,116,0,
                    "","",ATRIBUTO),
                    gotowindow(2),
                    str_len(ATRIBUTO,Long),
                    Long  0,!,
                    verifica(ATRIBUTO,REL,"",15,AA,BB),
                    gotowindow(2),
                    changestatus(
"F1 = AYUDA  ESC = TERMINAR   Digite los Atributos de la Relación y \17\217"),
                    concatena(ANTERIOR,[REL],FINAL),
                    X = AA + 1,
                    cursor(X,BB),
                    lee_atributos(ATR,FINAL).

lee_atributos(ATR,FINAL):-ATR = FINAL.

/* Verifica número de atributos dados de alta */

cambia(A,B):-A = 16,B  54,!,
        AA = A - 4,
        BB = B + 16,
        cursor(AA,BB).

cambia(A,B):-A = 16,B = 54,!,
        changestatus(
"F1 = AYUDA     ***   HASTA 16 ATRIBUTOS    ***      ESC = CONTINUAR"),
        sound(5,200),
        cursor(16,54),
        readkey(KEY).

cambia(A,B):-AA = A, BB = B,
        cursor(AA,BB).

/* Carga a la base los datos capturados */

almacena(REL,Atributos):- assertz(esquema(REL,Atributos)).


/* Lectura de Dependencias Funcionales */

leeDependencias:- inicia,
            changestatus(
"F1 = AYUDA      \17\217 = SELECCION          F10 = FIN  SELECCION          ESC = TERMINA
DEPENDENCIAS"),
            push_helpcontext(depende),
            esquema(NOMBRE,LISTA),not(LISTA = []),!,
            deps(NOMBRE,LISTA),
            pop_helpcontext,
            clearwindow.
leeDependencias.

inicia:-field_str(1,26,26," DEPENDENCIAS FUNCIONALES ").

deps(NREL,ATRIBS):-leeDep(LHS,RHS,ATRIBS),
            not(LHS = []),!,
```

```
            checa_1_atributo(NREL,LHS,RHS),
            deps(NREL,ATRIBS).

deps(_,_).

leeDep(LHS,RHS,LATR):-boxmenu_mult(6,7,4,64,63,30,LATR,
            " SELECCIONE LOS ATRIBUTOS DETERMINANTES",
            [],S1),
            inicia,
            sound(5,3500),
            invierte(S1,[],SEL1)
            not(SEL1 = []),!,
            recdeps(SEL1,LATR,[],LHS).
            clearwindow,
            inicia,
            boxmenu_mult(6,7,4,64,63,30,LATR,
            " SELECCIONE LOS ATRIBUTOS DETERMINADOS",
            [],S2),
            sound(5,3500),
            invierte(S2,[],SEL2).
            not(SEL2 = []),
            recdeps(SEL2,LATR,[],RHS),
            field_str(10,3,12,"DEPENDENCIA:"),
            cursor(12,3),
            imp_dep(LHS,RHS,12)

recdeps([S1|RS1],LISTA,ANT,DET):-not(RS1 = []),!,
                  obtiene(LISTA,S1,1,ATR),
                  concatena(ANT,[ATR],FDET),
                  recdeps(RS1,LISTA,FDET,DET)

recdeps([RS|RRS],LISTA,ANT,DET):-RRS = [],!,
                  obtiene(LISTA,RS,1,ATR),
                  concatena(ANT,[ATR],DET).


obtiene([_|RATR],ATRNO,CONT,Atributo):-not(ATRNO = CONT),!,
                        C = CONT + 1,
                        obtiene(RATR,ATRNO,C,Atributo).

obtiene([ATR|_],ATRNO,CONT,Atributo):-ATRNO = CONT,!,
                        Atributo = ATR


invierte([L1|L2],ANT,Linv):-not(L2 = []),!,
                  concatena([L1],ANT,LF),
                  invierte(L2,LF,Linv).

invierte([L1|L2],ANT,Linv):-L2 = [],!,
                  concatena([L1],ANT,Linv).

/* Transforma las dependencias en dependencias con un sólo
  atributo en la parte determinante */

checa_1_atributo(_,_,[]).
checa_1_atributo(NREL,LHS,[F|F1]):-concatena([F],[],R),
                        assertz(depfun(NREL,LHS,R)),
                        checa_1_atributo(NREL,LHS,F1).

/* Despliegado de información */

imprime_esquema:-clearwindow,
          esquema(A,B),not(B = []),!,
          esc_atrib("ESQUEMA DE LA RELACION : ",3,18,113),
          esc_atrib(A,3,43,116),
          str_len(A,LNRel),
```

```
            ORIGEN = LNRel + 1,
            field_str(5,1,LNRel,A),
            esc_lis(B,5,ORIGEN),
            continua.

imp_dependencias:-clearwindow,
            field_str(2,10,26,"DEPENDENCIAS FUNCIONALES :"),
            cursor(4,1),
            fds,!,
            continua.

fds:-depfun(_,LHS,RHS),
    cursor(RA,_),
    imp_dep(LHS,RHS,RA),
    cursor(RRA,_),
    verif(RRA,14),
    fail.

fds.

verif(RR,PARAM):-RR  PARAM,!.

verif(RR,PARAM):-RR = PARAM,!,
            continua,
            clearwindow,
            field_str(2,10,26,"DEPENDENCIAS FUNCIONALES :"),
            cursor(4,1).

/* Edición de Dependencias (Alta/Baja) */
opcion(0).

opcion(1):-lee_archivos("",NES,"ESQ"),
       not(NES = ""),
       lee_archivos("",NDE,"DEP"),
       not(NDE = ""),!,
       edita(1,NES,NDE).

opcion(2):-lee_archivos("",NDE,"DEP"),
       not(NDE = ""),!,
       edita(2,"",NDE).

opcion(_).

/* Control */

/* Lee Nombre de Archivos */

lee_archivos(NE,NVE,"ESQ"):-makewindow(3,78,78,"",4,4,3,73),
            changestatus(
"F1 = AYUDA \17\217 = Acepta Nombre      Edite nombre o presione \17\217"),
            field_str(0,0,31,
            "Nombre del archivo del Esquema"),
            push_helpcontext(archivos),
            readfilename(5,46,87,0,"ESQ",NE,NVE),!,
            refreshstatus,
            pop_helpcontext,
            removewindow.

lee_archivos(ND,NVD,"DEP"):-makewindow(3,78,78,"",4,4,3,73),
            changestatus(
"F1 = AYUDA \17\217 = Acepta Nombre      Edite nombre o presione \17\217"),
            field_str(0,0,35,
            "Nombre del archivo de Dependencias"),
            push_helpcontext(archivos),
            readfilename(5,46,87,0,"DEP",ND,NVD),!,
            refreshstatus,
            pop_helpcontext,
```

```
                removewindow.

lee_archivos(NBD,NVBD,"BDN"):-makewindow(3,78,78,"",4,4,3,73),
                changestatus(
"F1 = AYUDA  \17\217 = Acepta Nombre      Edite nombre o presione \17\217"),
                field_str(0,0,45,
                "Nombre del archivo de Datos de Normalizacion"),
                push_helpcontext(archivos),
                readfilename(5,50,116,0,"BDN",NBD,NVBD),!,
                refreshstatus,
                pop_helpcontext,
                removewindow.

lee_archivos(_,"",_):-!,removewindow.

/* Almacena en disco el esquema y las dependencias */

guarda:-esquema(A,_),
    checa_long(A,REL),
    concat(REL,".ESQ",NES),
    concat(REL,".DEP",NDE),
    lee_archivos(NES,NVE,"ESQ"),
    lee_archivos(NDE,NVD,"DEP"),
    not(NVE = ""),
    not(NVD = ""),!,
    almacena_archivo(1,NVE),
    almacena_archivo(2,NVD).

guarda:-!.

checa_long(REL,NREL):-str_len(REL,LONG),
            LONG8,!,
            frontstr(8,REL,NREL,_).
checa_long(REL,NREL):-!,NREL = REL.

almacena_archivo(1,NES):-writedevice(OLD),
                openwrite(aux,NES),
                writedevice(aux),
                wesq,
                closefile(aux),
                writedevice(OLD).

almacena_archivo(2,NDE):-writedevice(OLD),
                openwrite(aux,NDE),
                writedevice(aux),
                wdeps,
                closefile(aux),
                writedevice(OLD).

almacena_archivo(3,NDB):-writedevice(OLD),
                openwrite(aux,NDB),
                writedevice(aux),
                wesq,
                wdeps,
                wllave,
                wdecomp,
                wen3afn,
                winfd,
                closefile(aux),
                writedevice(OLD).

/* Formatea adecuadamente los datos capturados para almacenarlos */

wesq:-esquema(A,B),
   write("esquema(\"",A,"\",",B,")\13\10"),fail.
wesq.
wdeps:-depfun(A,B,C),
```

```
        write("depfun(\"",A,"\",",B,",",C,")\13\10"),fail.
wdeps.

wllave:-llave(A,B),
        write("llave(\"",A,"\",",B,")\13\10"),fail.
wllave.

wdecomp:-decomp(A,B),
        write("decomp(\"",A,"\",\"",B,"\")\13\10"),fail.
wdecomp.
wen3afn:-en3afn(A),
        write("en3afn(\"",A,"\")\13\10"),fail.
wen3afn.
wtnfd:-tnfdecomp(A,B),
        write("tnfdecomp(\"",A,"\",\"",B,"\")\13\10"),fail.
wtnfd.

/* Alta de dependencias */
/* Creación de un nuevo archivo, o bien agregar dependencias a un
   archivo existente */

edita(1,NE,ND):-not(existfile(NE)),
                no_existe(NE,
                " NO EXISTE EL ARCHIVO DEL ESQUEMA "),
                lee_archivos(NE,NVE,"ESQ"),
                not(NVE = ""),!,
                edita(1,NVE,ND).

edita(1,NE,ND):-existfile(ND),
                consult(NE),
                consult(ND),
                esquema(REL,_),
                depfun(REL,_,_),!,
                clearwindow,
                leeDependencias,
                almacena_archivo(2,ND).

edita(1,NE,ND):-not(existfile(ND)),!,
                consult(NE),
                clearwindow,
                leeDependencias,
                almacena_archivo(2,ND).

edita(1,NE,ND):-no_existe("",
" LAS DEPENDENCIAS Y EL ESQUEMA NO PERTENECEN A LA MISMA RELACION "),
                lee_archivos(NE,NVE,"ESQ"),
                lee_archivos(ND,NVD,"DEP"),
                not(NVE = ""),not(NVD = ""),!,
                edita(1,NVE,NVD).

edita(1,_,_):-!.

/* Baja de dependencias de algun archivo */

edita(2,"",ND):-not(existfile(ND)),
        no_existe(ND," NO EXISTE EL ARCHIVO DE DEPENDENCIAS "),
                lee_archivos(ND,NVD,"DEP"),not(NVD = ""),!,
                edita(2,"",NVD).

edita(2,"",ND):-consult(ND),
                depfun(R,A,B),
                clearwindow,
                imp_dep(A,B,5),
                cursor(18,50),
                field_str(18,39,10,"Borrar S/N"),
                readchar(T),
                versino(T,R,A,B),
```

```
            fail.

edita(2,"",ND):-almacena_archivo(2,ND),
            checa(ND),!.

edita(2,"",_):-!.

checa(ND):-not(ND = "").file_str(ND,A),
    A = "",!,
    deletefile(ND).
checa(_).

versino('S',REL,LHS,RHS):-!,retract(depfun(REL,LHS,RHS)).
versino('s',REL,LHS,RHS):-!,retract(depfun(REL,LHS,RHS)).
versino('N',_,_,_):-!.
versino('n',_,_,_):-!.
versino(_,REL,LHS,RHS):-!,sound(5,5000),
                sound(5,500),
                cursor(18,50),
                readchar(T),versino(T,REL,LHS,RHS).


no_existe(NOM,TXT):-!,sound(5,100),
            concat(TXT,NOM,ST),
            str_len(ST,LST),
            field_attr(18,3,LST,78),
            field_str(18,3,LST,ST).

/* Carga esquema a la base de datos */

carga(1):-borra_datos,
        lee_archivos("",NVE,"ESQ"),
        not(NVE = ""),
        carga1(1,NVE),!,
        clearwindow,
        menu(5,35,116,113,
     ["Esquema","Dependencias","Datos Normalizacion"],"",2,OP),
        carga(OP).

/* Carga dependencias a la base de datos */

carga(2):-borradep,
        lee_archivos("",NVD,"DEP"),
        not(NVD = ""),!,
        carga1(2,NVD).

/* Carga datos de normalización a la base */

carga(3):-borra_datos,
        lee_archivos("",NBD,"BDN"),
        not(NBD = ""),!,
        carga1(3,NBD).

carga(_):-!.



carga1(1,NOM):-existfile(NOM),!,
            consult(NOM),
            imprime_esquema.

carga1(1,NOM):-no_existe(NOM,
            " NO EXISTE ARCHIVO O NO CONTIENE ESQUEMA "),
            lee_archivos(NOM,NVO,"ESQ"),
            not(NVO = ""),!,
            carga1(1,NVO).
```

```
carga1(2,NOM):-not(existfile(NOM)),
        no_existe(NOM,
      " NO EXSISTE ARCHIVO O NO CONTIENE DEPENDENCIAS "),
        lee_archivos(NOM,NVO,"DEP"),
        not(NVO = ""),!,
        carga1(2,NVO).

carga1(2,NOM):-existfile(NOM),
        consult(NOM),
        esquema(A,_),
        depfun(A,_,_),!,
        imp_dependencias.

carga1(2,NOM):-no_existe("",
" EL ESQUEMA Y DEPENDENCIAS NO PERTENECEN A LA MISMA RELACION "),
        lee_archivos(NOM,NVO,"DEP"),
        not(NVO = ""),!,
        carga1(2,NVO).

carga1(3,NOM):-existfile(NOM),!,
        consult(NOM).

carga1(3,NOM):-no_existe(NOM,
      " NO EXSISTE ARCHIVO DE DATOS DE NORMALIZACION "),
        lee_archivos(NOM,NBD,"BDN"),
        not(NBD = ""),!,
        carga1(3,NBD).

carga1(_,_):-!.

/* Borra Datos cargados en la base */

borra_datos:-borraesq,
        borradep,
        borragpo,
        borrallave,
        borradecomp,
        borraen3afn,
        borrainfd.

borraesq:-esquema(_,_),
      retract(esquema(_,_)),
      fail.
borraesq.

borradep:-depfun(_,_,_),
      retract(depfun(_,_,_)),
      fail.
borradep.

borragpo:-grupo(_,_),
      retract(grupo(_,_)),
      fail.
borragpo.

borrallave:-llave(_,_),
       retract(llave(_,_)),
       fail.
borrallave.

borradecomp:-decomp(_,_),
       retract(decomp(_,_)),
       fail.
borradecomp.

borraen3afn:-en3afn(_),
       retract(en3afn(_)),
```

```
        fail.
borraen3afn.
borratnfd:- tnfdecomp(_,_).
        retract(tnfdecomp(_,_)),
        fail.
borratnfd

/* Despliega todos los datos cargados en la base*/

despliega:-makewindow(2,31,113,"",3,0,21,80),
        makewindow(4,30,30," Esquema Inicial ",3,0,6,80),
        makewindow(5,56,56,
        " Dependencias y Datos de Normalización ",9,0,15,80),
        despliega_esquema(REL),
        not(REL = ""),!,
        gotowindow(5),
        despliega_deps(REL),
        clearwindow,
        despliega_decomp(REL),quita_vent

despliega:-quita_vent.
quita_vent:-gotowindow(4),removewindow,
        gotowindow(5),removewindow,
        gotowindow(2),removewindow.

despliega_esquema(REL):-esquema(REL,ATR),
                not(decomp(_,REL)),!,
                gotowindow(4),
                esc_atrib("Esquema de la relación: ",
                0,5,30),
                esc_atrib(REL,0,30,31),
                esc_atrib(REL,1,3,30),
                str_len(REL,LREL),
                ORIGEN = 4 + LREL,
                esc_lis(ATR,1,ORIGEN).

despliega_esquema(REL):-not(esquema(_,_)),!,
                gotowindow(2),
                no_existe("NO EXISTE ESQUEMA CARGADO EN LA BASE",""),
                continua,
                REL = "".

despliega_deps(REL):-depfun(REL,_,_),!,
        field_str(0,10,26,"DEPENDENCIAS FUNCIONALES :"),
        cursor(1,1),
        fds,
        continua.

despliega_deps(_).

despliega_decomp(REL):-esquema(NOMBRE,ATRIBS),
                not(NOMBRE = REL),
                decomp(REL,NOMBRE),
        esc_atrib("DESCOMPOSICION EN TERCERA FORMA NORMAL DE",
                0,1,26),
                esc_atrib(REL,0,50,116),
                esc_atrib(NOMBRE,2,45,62),
                esc_atrib(NOMBRE,4,5,56),
                str_len(REL,L),
                C = 5 + L + 1,
                cursor(4,C),
                esc_lis(ATRIBS,4,C),
                imp_llaves(NOMBRE),
                fail.
despliega_decomp(_):-!.

/* Impresión de todos los datos cargados en la base */
```

```
impresion:-date(A,M,D),
        time(H,Min,S,HS),
 write("\t\tIMPRESION DE DATOS CARGADOS EN LA BASE DEL NORMALIZADOR.\n"),
 write("\nFECHA: ",D," /",M," /",A,"\n"),
 write("HORA:  ",H,":",Min,":",S,"\n"),
        imp_esq_orig(REL),
        imp_depfun(REL),
        imp_decomp(REL).

imp_esq_orig(REL):-esquema(REL,ATR),
        not(decomp(_,REL)),!,
 write("\tESQUEMA DE LA RELACION INICIAL: ",REL,"\n\n"),
        imp_esq(REL,ATR).

imp_depfun(REL):-depfun(REL,_,_),!,
 write("\t\t\tDEPENDENCIAS FUNCIONALES :\n\n"),
        ifds,write("\n\n").
imp_depfun(_).

imp_decomp(REL):-esquema(NOMBRE,ATRIBS),
        not(NOMBRE = REL),
        decomp(REL,NOMBRE),
 write("\nDESCOMPOSICION EN TERCERA FORMA NORMAL DE ",
        REL," EN ",NOMBRE,"\n\n"),
        imp_esq(NOMBRE,ATRIBS),
        write("\n"),
        llaves(NOMBRE),
        write("\n"),
        fail.
imp_decomp(_):-!.

imp_lista([],Origen,_,COL):-!,write("\n").

imp_lista([Cabeza|Resto],Origen,Limite,COL):-
                str_len(Cabeza,LL),
                NCOL = COL + LL + 1,
           (COL + LL + 1) Limite,!,
                write(Cabeza," "),
           imp_lista(Resto,Origen,Limite,NCOL).

imp_lista(Lista,Origen,Limite,COL):-write("\n"),
                        espacios(Origen),
           imp_lista(Lista,Origen,Limite,1).

espacios(0):-!.
espacios(ORG):-write(" "),
        NORG = ORG - 1,
        espacios(NORG).

ifds:-depfun(_,LHS,RHS),
   concatena(["( "],LHS,ILHS),
   concatena(ILHS,[") -- ("],NL),
   concatena(NL,RHS,DEP),
   concatena(DEP,[" )"],DEPFUN),
   imp_lista(DEPFUN,1,80,1),
   fail.

ifds.

imp_esq(REL,ATRIBS):-not(REL = ""),
           not(ATRIBS = []),
           write(REL),
           str_len(REL,L),
           ORIG = L + 2,
           concatena(["("],ATRIBS,ATR1),
           concatena(ATR1,[")"],ATR2),
```

```
                imp_lista(ATR2,ORIG,80,ORIG),
                write("\n").

llaves(REL):-llave(REL,LLAVE),
        concatena(["  Llave : "],LLAVE,LL),
        espacios(5),
        imp_lista(LL,5,80,5),
        fail.

llaves(REL).

/*  Fin Módulo interface */
/*************************************************************/
```

1.6 Manual de Operación.

El paquete normalizador ha sido elaborado utilizando la notación relacional, aunque su aplicación puede ser útil en cualquier modelo de datos.

Antes de utilizar el paquete, es necesario definir la relación que se desea normalizar, así como las dependencias funcionales entre atributos. Si la notación relacional no le es familiar, se puede entender a la relación como un registro y a los atributos, como el nombre de cada uno de los campos que conforman el registro. Las dependencias funcionales se pueden entender como las posibles dependencias entre campos que forman los identificadores existentes para el mismo registro.

Contenido del diskette:

El diskette del paquete normalizador contiene cuatro archivos, verifique la existencia de ellos antes de utilizar el programa.

Archivo	Descripción
NORMAL .EXE	Paquete normalizador
MENSAJES.CRT	Mensajes de ejecución
AYUDAS .DEF	Definición de ayuda en línea
AYUDAS .HLP	Definición de ayuda en línea

- Se recomienda que después de verificar la existencia de todos los archivos, se genere una copia de respaldo para no dañar el disco original.

Como entrar.

Posicionarse en el disco y directorio en donde se encuentra el paquete normalizador y teclear lo siguiente:

› NORMAL<Enter >

Como salir.

En el menú principal oprima la tecla T, o posicionar el cuadro cursor sobre la palabra Terminar y presionar Enter.

El paquete normalizador presenta un menú principal que ofrece las siguientes cuatro opciones:

a) Edición.

b) Base de Datos.

c) Normalización.

d) Terminar.

Cada opción contiene a su vez submenúes, que van guiando al usuario a través del proceso de normalización de una manera amigable y fácil de usar. Antes de describir cada una de las opciones, mencionamos a continuación las generalidades aplicables a lo largo de la utilización del paquete normalizador.

Selección de opción en menúes.

Para seleccionar una opción de cualquier menú, basta con oprimir la primera letra de cada opción, o bien, posicionar el cuadro cursor sobre la opción deseada mediante las flechas y presionar Enter.

Para cancelar cualquier opción seleccionada se debe oprimir la tecla Esc.

Ayudas de contexto.

El paquete normalizador proporciona ayuda de contexto que se puede activar en cualquier momento, mediante la tecla de función F1.

Linea de estatus.

En la parte inferior de la pantalla aparece una linea de estatus que indica los mnemónicos de las teclas de control, así como una breve descripción de la acción que cada tecla realiza al ser presionada.

Nombres de archivos.

Las convensiones utilizadas en el paquete normalizador, sugieren tres tipos de extensiones para almacenar la información:

Archivos de esquemas : XXXXXXXX.ESQ

Archivos de dependencias : XXXXXXXX.DEP

Archivos de base de datos del normalizador : XXXXXXXX.BDN

Cuando se solicitan nombres de archivos, en ocasiones aparece un nombre por omisión asignado por el paquete, al oprimir Enter se acepta el nombre. Sin embargo, si se desea modificar ese nombre, se utilizan las teclas de edición como son: flecha, inicio, fin, borrado, backspace, etc. para modificar el nombre. No es necesario digitar la extensión del archivo.

Si se oprime la tecla Enter con el nombre del archivo en blanco, se desplegará una ventana mostrando los archivos existentes en el directorio actual.

a) Edición.

Esta opción permite realizar dos funciones:

Edición de esquema.

Esta constituye el primer paso en el proceso de normalización.

Al seleccionar la edición de esquema, aparecen las pantallas que permiten dar de alta un esquema junto con un conjunto de dependencias funcionales.

Primero se solicita el nombre de la relación. Una vez tecleado y oprimiendo Enter, se solicitan los los nombres de los atributos que forman el esquema de dicha relación.

Los nombres de esquemas y atrubutos deben de cumplir un sintaxis mínima para que le paquete normalizador pueda operar satisfactoriamente. Esta sintaxis requiere únicamente de lo siguiente:

- Los nombres siempre deben iniciar con letra, seguida de cualquier secuencia de letras, números y el carácter de subrayado.
- La longitud del nombre del esquema debe ser cuando más de 20 carácteres.
- La longitud del nombre de los atributos debe ser cuando más de 15 carácteres.

El sistema valida la sintaxis de los nombres digitados y limita la longitud de los nombres según lo mencionado anteriormente.

El número máximo de atributos por relación es de 16.

Una vez capturados todos los atributos, se debe terminar la captura mediante la tecla Esc, tal como lo indica la linea de estatus, o presionando Enter en un renglón en blanco.

Inmediatamente después aparece el esquema y todos los atributos capturados para proceder a capturar las dependencias funcionales de los atributos.

El procedimiento de captura de las dependencias funcionales es muy sencillo gracias a que todos los atributos se presentan en pantalla para facilitar la selección de los mismos.

Este procedimiento consiste de los siguientes pasos:

1) Seleccionar los atributos determinantes de la dependencia es decir, la parte izquierda de la dependencia funcional (la llave).

   Para seleccionar los atributos, se debe posicionar el cursor en el atributo deseado y oprimir la tecla Enter. Oprimiendo nuevamente la tecla Enter el atributo pierde la propiedad de selección. Un atributo seleccionado se reconoce porque aparece resaltado en video inverso.

Cabe aclarar que para efectos del normalizador el orden de selección de atributos debe ser el mismo con el que fueron capturados en el esquema. Es por ésto que el movimiento dentro de la ventana de selección sólo se permite de izquierda a derecha y de arriba hacia abajo. Invariablemente, la selección de atributos debe realizarce por renglón, de izquierda a derecha.

Una vez capturados los atributos determinantes, se debe oprimir la tecla de función F10 para indicar que los atributos determinantes han sido seleccionados por completo.

2) Selección de los atributos determinados, es decir, los atributos de la parte derecha de la dependencia.

El proceso de selección es el mismo que para los atributos determinantes.

Si la parte determinada no contiene atributos, ésta no será grabada en el archivo de dependencias.

Al terminar de capturar todas las dependencias funcionales, es necesario oprimir la tecla Esc para indicar que se han dado de alta todas las dependencias funcionales asociadas al esquema de la relación.

Seguido de la captura de dependencias funcionales se presentan todas las dependencias definidas con un sólo atributo en la parte determinada, esto es para facilitar la operación interna del paquete normalizador, lo cual no afecta a la definición original.

El normalizador solicita entonces el nombre del archivo que se utilizará tanto para almecenar el esquema, con extensión .ESQ, así como las dependencias funcionales, con extensión .DEP.

Se toman los ocho primeros carácteres del nombre de la relación para integrar los nombres de ambos archivos, Sin embargo se pueden modificar al oprimir cualquier tecla diferente del Enter. Si se desea almacenar la información capturada con los nombres por omisión, sólo basta oprimir Enter. Si se desean modificar los nombres, se pueden editar oprimiendo cualquier tecla diferente, utilizando backspace, flechas etc.. Si se desea

observar una lista de archivos del directorio actual, se debe oprimir Enter con el nombre en blanco.

Una vez almacenados el esquema y las dependencias funcionales, el cursor regresará al menú principal.

Edición de dependencias funcionales.

Para facilitar la iteratividad del proceso de normalización, esta opción permite agregar y eliminar dependencias a un archivo ya existente, o bien, permite dar de alta un nuevo archivo de dependencias asociado a un esquema dado.

Al seleccionar esta opción aparece un menú que permite realizar dos funciones:

- Alta de dependencias.
- Baja de dependencias.

Alta de dependencias (crear un archivo nuevo de dependencias o agregar dependencias a un archivo existente)

Para el caso de Alta de dependencias, se solicita el nombre del archivo tanto del esquema como de las dependencias.

Si el archivo de dependencias que se digita no existe, crea un nuevo archivo de dependencias asociado al archivo del esquema. Si el archivo del esquema no existe, se marca un error, ya que siempre se debe asociar un esquema con sus dependencias.

Si el archivo de dependencias digitado existe en el directorio actual, las dependencias que se den de alta en esta opción serán agregadas a este archivo.

Baja de dependencias (eliminar dependencias de un archivo)

Seleccionando esta función se solicita el nombre del archivo de dependencias, mostrando en pantalla una por una todas las dependencias del esquema y se pregunta si se desea borrar.

La facilidad de iteratividad consiste en tener un archivo de esquema y varios archivos de dependencias funcionales, ya que el proceso de normalización dará diferentes resultados y permitirá dar el significado real a la información al comparar diversos resultados, producto de diversos archivos de dependencias funcionales.

b) Base de Datos.

Esta opción permite realizar varias funciones para manejar la información de la base de datos del normalizador, entre las cuales encontramos las siguientes:

- Carga
- Almacena en Disco
- Despliega
- Borra
- Imprime

La primera función es la de carga de datos, la cual permite cargar a la base tanto esquema como dependencias funcionales antes de iniciar la normalización. Es importante que el archivo de dependencias y el del esquema pertenezcan a la misma relación pues de lo contrario se marcará un error.

Para la selección de nombres de archivos se utilizará el manejo de pantallas anteriormente descrito.

La carga del esquema y de las dependencias se selecciona mediante un menú, de manera independiente para permitir asociar diferentes conjuntos de dependencias a un mismo esquema.

La función de almacena en disco permite guardar todos los datos del esquema, de las dependencias y los datos de normalización, en un archivo para posteriores consultas, impresiones y principalmente para facilitar la comparación de resultados.

La función despliega permite observar todos los datos cargados en la base; nuevamente se pueden observar tanto esquema como dependencias y datos de normalización (nuevas relaciones generadas al normalizar, etc).

La función imprimir presenta un reporte impreso de todos los datos cargados en la base.

c) Normalización.

Dentro de esta opción, se tienen dos modos de operación:

- Tutorial y
- Directo

El modo tutorial se enfoca hacia los usuarios interesados en entender los algoritmos de normalización de Bernstein, ya que aquí se explica paso a paso el proceso de normalización que lleva a obtener uno u otro resultado, normaliza tomando como entrada el esquema y dependencias que se tienen en ese momento cargadas en la base de datos.

El modo directo permite obtener resultados inmediatos. Para ambos casos, se muestra al final como resultado de la normalización, las relaciones generadas en este proceso así como sus respectivos esquemas y las llaves asociadas a cada uno de ellos.

Al terminar la normalización, todos los datos se mantienen cargados en la base. Si se desea almacenar el resultado, hay que hacer uso de la opción almacena en disco de base de datos o bien si se desea normalizar con un conjunto diferente de dependencias hay que cargar un nuevo conjunto de dependencias asociado al esquema.

d) Terminar.

Para terminar la sesión.

Secuencia de Operación.

A continuación se presenta un bosquejo general de la secuencia que sugerimos para utilizar el normalizador de forma tal que proporcione información útil al diseño y al entendimiento de la Teoría de Normalización.

Primera Vez:

1) Edición de Esquema

Dar de alta Esquema y Dependencias.
(Los datos quedan cargados en la base)

2) Normalización

3) Almacenar resultados y/o Imprimir resultados

Subsecuentes:

1) Edición de Dependencias

Dar de alta nuevos conjuntos de dependencias para el esquema.

2) Cargar Esquema y Dependencias a la Base de Datos del Normalizador.

3) Normalizar

4) Analizar y/o Imprimir datos

5) Realizar punto 2).

6) Analizar los diferentes Resultados.

Glosario.

A continuación se presenta un glosario de términos utilizados en el paquete normalizador en la modalidad tutorial para facilitar el entendimiento de éste, de manera práctica y rápida.

Atributo Extraño.

Atributo que se encuentra en la parte determinante de una dependencia funcional y que no es necesario para determinar el lado derecho de la dependencia. Una dependencia parcial contiene atributos extraños ya que un atributo determinado depende parcialmente de la llave.

Closure.

Se obtiene el closure de la parte determinante de una dependencia con respecto a un conjunto de dependencias, para saber todos los atributos que esa parte determina, útil para detectar dependencias parciales y transitivas.

Dependencia funcional redundante.

Dependencia que no proporciona información porque su parte determinada se obtiene del conjunto de dependencias sin incluirla. Una dependencia transitiva se traduce en una dependencia funcional redundante.

Dependencia parcial.

Cuando los atributos determinados dependen de una parte de la llave (parte determinante).

Dependencia transitiva.

Cuando existe $A \longrightarrow B$ y $B \longrightarrow C$.

Esquema.

Asocia una relación con sus atributos.

Parte determinada.

Lado derecho de una dependencia funcional.

Parte determinante.

Lado izquierdo de una dependencia funcional. Llave supuesta de un registro.

Llaves equivalentes.

Partes determinantes diferentes cuyas partes determinadas son iguales.

Grupo.

Asocia una parte determinante con todos los atributos que ésta determina.

1.7 Pruebas.

Como elemento práctico del presente trabajo se muestra el reporte impreso de la ejecución del normalizador considerando los siguientes ejemplos:

Ejemplo 1.

Considere la siguiente relación:

Nombre : Conciertos

Atributos :

- Instrumento
- Obra
- Intérprete
- Sala
- Estilo
- Autor

Esta relación contiene información de información existente en determinada Asociación de Música referente a conciertos, tipos de instrumentos, intérprete, sala, estilo, obra y autor, considerando las siguientes condiciones de dependencia:

a) Intérprete depende del instrumento y la obra. Esto es, para hacer referencia a un intérprete específico se necesita saber qué instrumento domina y qué obra puede o tiene asignada para interpretar.

b) El intrumento depende del intérprete que lo toque. Esto es, para saber que instrumento se domina dentro de la Asociación es necesario saber el nombre del interprete que lo puede dominar.

c) El autor depende del estilo y la sala. Esto limita el alcance de la difusión del autor, ya que el autor está limitado a ser interpretado en aquellas salas que estén consideradas de algún estilo definido.

d) La sala depende del estilo. Las salas tocan únicamente cierto estilo. Además el estilo a su vez depende de sala pues determinado estilo es interpretado solo en salas selectas.

e) El autor depende de la sala, pues se tiene clasificada la información de los autores por el estilo en que componen.

Dependencias de entrada :

Instrumento, Obra ——→ Intérprete.

Intérprete ——————→ Instrumento.

Sala, Estilo ——————→ Autor.

Estilo ——————→ Sala.

Estilo ——————→ Autor.

Sala ——————→ Estilo.

IMPRESION DE DATOS CARGADOS EN LA BASE DEL NORMALIZADOR.

FECHA: 23 /1 /1989

HORA: 21:19:4

ESQUEMA DE LA RELACION INICIAL: Conciertos

Conciertos(Instrumento Obra Interprete Sala Estilo Autor)

DEPENDENCIAS FUNCIONALES :

(Sala) -> (Estilo)

(Estilo) -> (Autor)

(Estilo) -> (Sala)

(Interprete) -> (Instrumento)

(Instrumento Obra) -> (Interprete)

DESCOMPOSICION EN TERCERA FORMA NORMAL DE Conciertos EN Conciertos_a

Conciertos_a(Estilo Autor)

Llave : Estilo

Llave : Sala

DESCOMPOSICION EN TERCERA FORMA NORMAL DE Conciertos EN Conciertos_b

Conciertos_b(Instrumento Obra Interprete)

Llave : Instrumento Obra

DESCOMPOSICION EN TERCERA FORMA NORMAL DE Conciertos EN Conciertos_c

Conciertos_c(Instrumento Interprete)

Llave : Interprete

Ejemplo 2.

Considere la siguiente relación:

Nombre : Expertos

Atributos :

- Número de empleado
- Nombre
- Experto en
- Edad
- Localización

Esta relación contiene información respecto a la localización de empleados expertos en alguna disciplina considerando semánticamente lo siguiente:

a) Cada empleado tiene un número asociado único.

b) Debido a los requerimientos geográficos de la compañia la localización determina la disiplina que domina cada empleado.

c) Todos los empleados tiene un localización única.

d) Todos los empleados son expertos en una sóla disiplina.

Dependencias de entrada:

NUM_EMP ——— NOMBRE_EMP,
EXPERTO_EN,
EDAD,
LOCALIZACION.

LOCALIZACION —— EXPERTO_EN.

IMPRESION DE DATOS CARGADOS EN LA BASE DEL NORMALIZADOR.

FECHA: 23 /1 /1989

HORA: 21:22:21

ESQUEMA DE LA RELACION INICIAL: EXPERTOS

EXPERTOS(NUM_EMP NOMBRE_EMP EXPERTO_EN EDAD LOCALIZACION)

DEPENDENCIAS FUNCIONALES :

(LOCALIZACION) -> (EXPERTO_EN)

(NUM_EMP) -> (LOCALIZACION)

(NUM_EMP) -> (EDAD)

(NUM_EMP) -> (NOMBRE_EMP)

DESCOMPOSICION EN TERCERA FORMA NORMAL DE EXPERTOS EN EXPERTOS_a

EXPERTOS_a(NUM_EMP NOMBRE_EMP EDAD LOCALIZACION)

Llave : NUM_EMP

DESCOMPOSICION EN TERCERA FORMA NORMAL DE EXPERTOS EN EXPERTOS_b

EXPERTOS_b(EXPERTO_EN LOCALIZACION)

Llave : LOCALIZACION

Ejemplo 3.

Tomando el ejemplo del Capítulo cinco, tenemos las siguientes dependencias de entrada:

EMP ——→ POS,

EXP,

SDO.

EMP,LNG —→ USO.

EMP,PRY —→ JFE.

IMPRESION DE DATOS CARGADOS EN LA BASE DEL NORMALIZADOR.

FECHA: 23 /1 /1989

HORA: 21:20:50

ESQUEMA DE LA RELACION INICIAL: PERSONAL

PERSONAL(EMP POS EXP SDO LNG USO PRY JFE)

DEPENDENCIAS FUNCIONALES :

(EMP PRY) -> (JFE)

(EMP LNG) -> (USO)

(EMP) -> (SDO)

(EMP) -> (EXP)

(EMP) -> (POS)

DESCOMPOSICION EN TERCERA FORMA NORMAL DE PERSONAL EN PERSONAL_a

PERSONAL_a(EMP POS EXP SDO)

Llave : EMP

DESCOMPOSICION EN TERCERA FORMA NORMAL DE PERSONAL EN PERSONAL_b

PERSONAL_b(EMP LNG USO)

Llave : EMP LNG

DESCOMPOSICION EN TERCERA FORMA NORMAL DE PERSONAL EN PERSONAL_c

PERSONAL_c(EMP PRY JFE)

Llave : EMP PRY

III
Conclusiones

Conclusiones.

A lo largo del presente trabajo encontramos que:

1) Existen grandes deficiencias en cuanto al proceso de diseño de sistemas de cómputo.

2) Cada vez más se reconoce a la información como un recurso importante para la toma de decisiones oportunas, esto es: "el poder lo tiene quien tiene la información y no quien tiene el dinero".

3) El proceso de diseño es empírico, ya que no existe una metodología que garantice un buen diseño, por lo contrario, observamos divergencias entre los diferentes autores, donde los únicos puntos en común son la intuición e iteratividad de dicho proceso.

4) El proceso de normalización utiliza conceptos abstractos que dificultan su comprensión.

5) La normalización depende fuertemente de cómo el diseñador interpreta los datos y sus relaciones.

6) El gran impacto teórico que ha tenido la Teoría de Normalización se debe a que ésta introdujo por primera vez conceptos matemáticos en lo que se refiere al manejo de datos.

7) Resulta difícil llevar a la práctica los conceptos contenidos en la Teoría de la Normalización debido a su nivel de abstracción.

8) Existe confusión en cuanto a los alcances de la Inteligencia Artificial.

9) PROLOG es un lenguaje que facilita la programación de procesos altamente recursivos.

En cuanto al paquete normalizador encontramos que:

10)Permite utilizar la Teoría de Normalización sin conocerla a fondo.

11)Presenta un modo tutorial que coadyuva al entendimiento del proceso de Normalización.

12)Minimiza las relaciones generadas ya que se identifican atributos irrelevantes.

13)Garantiza que se lleven a cabo todos los pasos requeridos en el proceso de Normalización evitando errores.

14)Contribuye a esclarecer las relaciones entre datos.

15)Por ser una herramienta interactiva es posible modificar fácilmente las dependencias para apoyar la naturaleza cíclica del diseño.

Adicionalmente encontramos que:

- La profundidad misma de cada tema impide una cobertura total en un sólo trabajo, esto nos llevó a reestructurar el planteamiento original, que pretendía cubrir tópicos fuera de contexto, lo que sugirió extraer los temas necesarios para abarcar los conceptos fundamentales en torno al Paquete Normalizador.

- La rápida evolución de los sistemas de información nos permitió reconocer la importancia que éstos tienen hoy en día, puesto que al obtener información veraz y oportuna, se tienen los elementos para una mejor toma de decisiones que influyan en el medio de una manera favorable a las organizaciones.

- No es fácil mecanizar el pensamiento humano, pero si es posible mecanizar el proceso de normalización.

- Los paquetes desarrollados con una interface amigable, estructurados en base a menúes y complementados con una ayuda de contexto promueven la aceptación de los usuarios.

- Al utilizar el paquete normalizador, confirmamos la importancia de la semántica de los datos. Aún entendiendo el significado de los mismos, nos enfrentamos a resultados inesperados. Bajo la modalidad tutorial, descubrimos que en ocasiones se eliminaban dependencias aparentemente redundantes, debido a una identificación incorrecta de las dependencias funcionales.

- La limitación más importante para la mecanización del proceso de diseño de sistemas de información en general, es la falta de conocimiento para poder implantar herramientas que analicen semánticamente los datos.

- La Inteligencia Artificial ha hecho aportaciones importantes. Como ejemplo de ello tenemos la programación orientada a objetos, que abrió el espectro de la programación tradicional al obtener un enfoque diferente que pretende acercarse al razonamiento humano.

- Observamos que la teoría siempre va un paso adelante de la tecnología, por esto, en un futuro no muy lejano se tendrán sistemas que utilicen el conocimiento, razonamiento y procesamiento simbólico conjunta y eficientemente para modelar el pensamiento humano.

IV
Referencias

Referencias

A. Armstrong W. W., DEPENDENCY STRUCTURES OF DATABASE RELATIONS, en Information processing. North-Holland Publishing Co, Amsterdam Holanda, 1974.

B. Bernstein P. A., SYNTHESIZING THIRD-NORMAL-FORM RELATION FROM FUNCTIONAL DEPENDENCIES. ACM Transactions on Database Systems, 1, 1, diciembre 1976.

C. Blanning R. W., ISSUES IN THE DESIGN OF RELATIONAL MODEL MANAGEMENT SYSTEMS, en Proceedings of the National Computer Conference. AFIPS Press, junio 1983.

D. Borland International Inc., TURBO PROLOG USER MANUAL, 2da edición.

E. Bouzeghoub M., Gardain G. and Metais E., DATABASE DESIGN TOOLS: AN EXPERT SYSTEM APROACH, en Proceedings of the 11th VLDB Estocolmo, Suecia, agosto 21 1985.

F. Cárdenas Alfonso F., SISTEMAS DE ADMINISTRACION DE BANCO DE DATOS, editorial Limusa 1982.

G. Ceri S., Gottlob G., NORMALIZATION OF RELATIONS AND PROLOG, Communications of the ACM. 29, 6, noviembre 1986.

H. Clark K.L., Mc Cabe F.G., MICRO-PROLOG: PROGRAMMING IN LOGIC, Prentice-Hall International Series, 1984.

I. Cood E. F., FURTHER NORMALIZATION OF THE DATABASE RELATIONAL MODEL, en Data Base Systems Courant Computer Science Symposia. Vol. 6, editorial Prentice-Hall 1972.

J. Codd E. F., A RELATIONAL MODEL OF DATA FOR LARGE SHARED DATA BANKS, Communications of the ACM 13, 6, junio 1970.

K. Cullinet Software Inc., IDMS ARCHITECT USER MANUAL.

L. Cullinet Software Inc., DATABASE DESIGN AND DEFINITION GUIDE.

M. Date C. J., AN INTRIDUCTION TO DATABASE SYSTEMS, Vol II. Adisson-Wesly Publishing Co.

N. Date C. J., AN INTRODUCTION TO DATABASE SYSTEMS, 2da edición, Addison-Wesly Publishing Co. 1977.

O. Fagin R., FUNCTIONAL DEPENDENCIES IN A RELATIONAL DATABASE AND PROPOSITIONAL LOGIC, IBM J. Res. Dev. 21, 6, noviembre 1977.

P. Fairley Richard, INGENIERIA DE SOFTWARE, Mc Graw-Hill de México 1987.

Q. Floid L. Ruch y Philipe G. Zimbardo, PSICOLOGIA Y VIDA, E Editorial Trillas México 1979.

R. James O. Whittaker, PSICOLOGIA, editorial Interamericana 1971.

S. Krajewski R., DATABASE TYPES, Byte octubre 1984.

T. Kroenke David M., DATABASE PROCESSING FUNDAMENTALS DESIGN IMPLEMENTATION, Science Research Associates Inc. Estados Unidos de América 1983.

U. Pressman Roger S., SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, Mc Graw-Hill International Book Company 1982.

V. Revista 010 Vol.7 #9 mayo 1987.

W. Rich Elaine, ARTIFICIAL INTELLIGENCE, Mc Graw-Hill International Book Company 1983.

X. Samuel A. L., SOME STUDIES IN MACHINE LEARNING USING THE GAME OF CHECKERS, Mc Graw-Hill 1963.

Y. Wiederhold Gio., DISEÑO DE BASES DE DATOS, Mc Graw-Hill de México 1985.

Z. Yourdon Edward, Constantine Larry, STRUCTURED DESIGN, Prentice-Hall 1979.

V
Bibliografía

Bibliografía

Aho A.V., Beeri C., Ullman J.D. , THE THEORY OF JOINS IN RELATIONAL DATABASES, ACM Transactions on Database Systems 4, 3 septiembre 1979.

Beeri C., ON THE MEMBERSHIP PROBLEM FOR FUNCTIONAL AND MULTIVALUED DEPENDENCIES IN RELATIONAL DATABASES, ACM Transactions on Database Systems 5, 3, septiembre 1980.

Beeri C., Bernstein P.A., COMPUTATIONAL PROBLEMS RELATED TO THE DESIGN OF NORMAL FORM RELATIONAL SCHEMATA, ACM Transactions on Database Systems 4, 1, marzo 1979.

Beeri C., Bernstein P.A., Goodman N.A, SOPHISTICATE'S INTRODUCTION TO DATABASE NORMALIZATION THEORY, en Proceedings of the 4th International Conference On Very Large Data Bases, Alemania Oriental 1978.

Bobrow Daniel G., Mittal Sanjoy, Stefik Mark J., EXPERT SYSTEMS: PERILS AND PROMISE, Communications of the ACM, Vol 29 No 9 septiembre 1986.

Cuadrado Clara, Cuadrado John, PROLOG GOES TO WORK, Byte agosto 1985.

Cullinet Software Inc. CULLINET SYSTEM SOFTWARE GLOSSARY.

Eisenbach Susan, Sadler Chris, DECLARATIVE LANGUAJES: AN OVERVIEW, Byte agosto 1985.

Ferguson Ron, PROLOG: A STEP FORWARD THE ULTIMATE COMPUTER LANGUAGE, Byte, noviembre 1981.

Fisher Edward M, BUILDING A. I. BEHIND CLOSED DOORS, Datamation, Vol 32, No 15, agosto 1986.

Gagle M., Koehler G. J., Whinston A., DATA-BASE MANAGEMENT SYSTEMS: POWERFUL NEWCOMERS TO MICRCOMPUTERS, Byte noviembre 1981.

Gottlob. G., COMPUTING COVERS FOR EMBEDDED FUNCTIONAL DEPENDENCIES, Intern. Rep. 86-006, Dipartimento di Elettronica, Politecnico di Milano, Italia.

Jimenez Alejandro, PORQUE PROLOG, 0 1 0, Vol 7 No. 9 mayo 1987.

Kershberg L. Ed., EXPERT DATABASE SYSTEMS, en Proceedings of the 1st International Conference On Expert Database Systems Islas Kiawah S. C. octubre 1984.

Konopasek Leilos, Jayaraman Sundareson, EXPERT SYSTEMS FOR PERSONAL COMPUTERS, Byte mayo 1984.

Kowalski Robert, LOGIC PROGRAMMING, Byte agosto 1985.

Lee R. M., DATABASE INFERENCING FOR DECISION SUPPORT, Decision Support systems 1, 1, enero 1985.

Loizou G., Thanish P., TESTING A DEPENDENCY-PRESERVING DECOMPOSITION FOR LOSSLESSNESS, Infomation Systems 8, 1 marzo 1983.

Lucchesi C. L., Osborn S. L., CANDIDATE KEYS FOR RELATIONS, J. Comput. Syst. Sci. 17, 2, octubre 1978.

Marcot Bruce, TESTING YOUR KNOWLEDGE BASE, A. I. Expert, julio 1987.

Odette Lou, HOW TO COMPILE PROLOG, A. I. Expert, agosto 1987.

Potter W. D., DESIGN-PRO: A MULTI-MODEL SCHEMA DESIGN TOOL IN PROLOG, en Proceedings of the 1st International Conference on Expert Database Systems. Islas Kiawah, S. C. octubre 1984.

Salzberg Steven, KNOWLEDGE REPRESENTATION IN THE REAL WORLD, A.I. Expert, julio 1987.

Simons Gary F., DATA ABSTRACTION, Byte octubre 1984.

Tsou D. M., Fischer P. C., DECOMPOSITION OF A RELATION SCHEME INTO BOYCE-CODD NORMAL FORM, ACM-SIGACT 14, 3, verano 1982.

Yu C. T., Johnson D. T., ON THE COMPLEXITY OF FINDING THE SET OF CANDIDATE KEYS FOR A GIVEN SET OF FUNCTIONAL DEPENDENCIES, Inf. Process. Let 5, 4, octubre 1976.

Zaniolo C., Melkanoff M. A., ON THE DESIGN OF RELATIONAL DATABASE SCHEMATA, ACM Transactions on Database Systems 6, 1, marzo 1981.