Capítulo 4.
Lenguajes de bases de datos

Este capítulo describen los distintos lenguajes para bases de datos, la forma en que se puede escribir un lenguaje de consultas, se explican los dos enfoques: la compilación y la interpretación, y se mencionan herramientas disponibles para su construcción.

Las operaciones relaciónales tales como la *selección, proyección y reunión* son operaciones a nivel de conjuntos. Como consecuencia se dice que los lenguajes de bases de datos tales como SQL son llamados lenguajes *no-procedurales* , con la idea de que el usuario especifica ¿que?, no ¿como?. El proceso de navegar alrededor de la base de datos para dar respuesta a la solicitud del usuario es ejecutada automáticamente por el sistema, no manualmente por el usuario. Decidir como ejecutar esta navegación automática es responsabilidad de un elemento muy importante en un SMBD llamado el *optimizador* [Date 1995].

Diversas son las operaciones que se pueden llevar a cabo sobre una base de datos. De acuerdo al tipo de operaciones que puede realizar el lenguaje, estos pueden clasificarse como: *lenguajes de consultas, lenguajes de manipulación de datos (DML) y lenguajes de definición de datos (DDL)* principalmente. Ver figura 4.1.

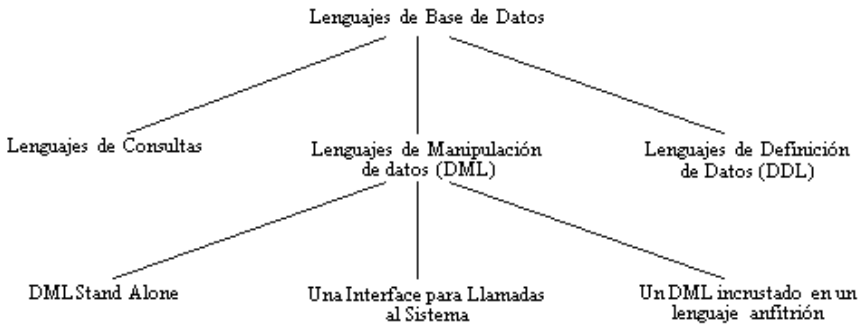

Figura 4.1 Clasificación de los lenguajes de bases de datos

4.1 Lenguaje de Consultas

Es un lenguaje de *no-programación* en el cual un usuario puede formular consultas y posiblemente también actualizar la base de datos.

No-programación significa que el usuario no tiene que especificar un algoritmo para obtener resultados, sino solamente definir la consulta de una manera ordenada.

4.2 Lenguaje de Manipulación de Datos

Es un lenguaje de programación que tiene una capacidad poderosa de calculo, flujo de control, entrada-salida, también tiene constructores sintácticos para acceso a base de datos (actualización, recuperación e intercambio dinámico de datos entre el programa y la base de datos). El DML es utilizado por el programador de la aplicación.

Un lenguaje de manipulación de datos puede ser:

Un DML stand-alone. En este caso el SMBD provee de un compilador o interprete para el DML. La desventaja de este lenguaje es que no puede ser usado para programas complejos, los cuales ejecutan algún acceso a la base de datos, pero simultáneamente ejecutan otras tareas, por ejemplo, cálculos numéricos.

Una Interface para Llamadas al Sistema. El usuario escribe un programa en un lenguaje de programación tradicional. El usuario ejecuta accesos a la base de datos por llamadas a subrutinas al SMBD. Las llamadas al sistema son interpretadas en tiempo de ejecución del programa. Una desventaja es que si la llamada al sistema contiene una solicitud incorrecta, el usuario no puede ser notificado en tiempo de compilación, sino que tiene que esperar hasta que el programa aborte.

Un DML Incrustado en un Lenguaje de Programación Anfitrión. Este es una extensión de acceso a base de datos de un lenguaje de programación de propósito general. El SMBD precompila el programa en un programa en el lenguaje anfitrión sin las sentencias del DML. Durante la precompilación el SMBD valida la sintaxis y la compatibilidad con el esquema de la base de datos. El SMBD puede también ejecutar optimización del algoritmo del usuario.

El programa resultante es compilado por el compilador del lenguaje anfitrión. Cuando el programa se ejecuta, este puede comunicarse con el SMBD, pero las llamadas al sistema de esta comunicación son transparentes al usuario.

4.3 Lenguaje de Definición de Datos

Es un lenguaje, en el cual la estructura lógica de la información puede ser definida, junto con su interpretación pragmática para el manejo de una base de datos, incluyendo el esquema, restricciones de integridad y vistas de usuario.

4.4 Construcción de Lenguajes de Bases de Datos

La tecnología de los compiladores, así como de los interpretes ha sido utilizada para el desarrollo de lenguajes de bases de datos. La compilación es ciertamente ventajosa desde el punto de vista de desempeño, esta tiene casi siempre un mejor desempeño en tiempo de ejecución que la interpretación [Donald 1981] (mencionado en [Date 1995]). Sin embargo Date [1995] dice que esto tiene un desventaja significante: "es posible que decisiones hechas por el compilador en tiempo de compilación no tengan validez larga en tiempo de ejecución". En la siguiente sección se describen los dos enfoques.

4.5 Compiladores

Las primeras tres fases suelen agruparse en una sola fase llamada *fase de análisis* y las últimas tres en una llamada *fase de síntesis*. La fase de análisis y el modulo de manejo de errores se describen posteriormente en este mismo capítulo. La fase de síntesis no es relevante en el contexto de un lenguaje multibase de datos, ya que este sigue un enfoque diferente que el de los lenguajes tradicionales, por esta razón solo se menciona.

Muchas herramientas de software que manipulan programas fuente realizan primero algún tipo de análisis, entre estas se encuentran los editores de estructuras, impresoras estéticas, verificadores estáticos y los interpretes [Aho et al. 1990].

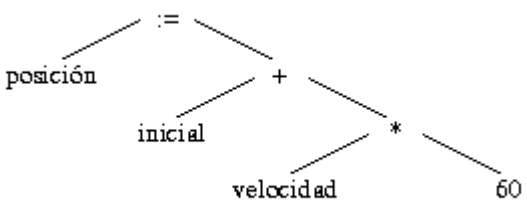

Figura 4.2 Árbol sintáctico para
posición:= inicial + velocidad * 60

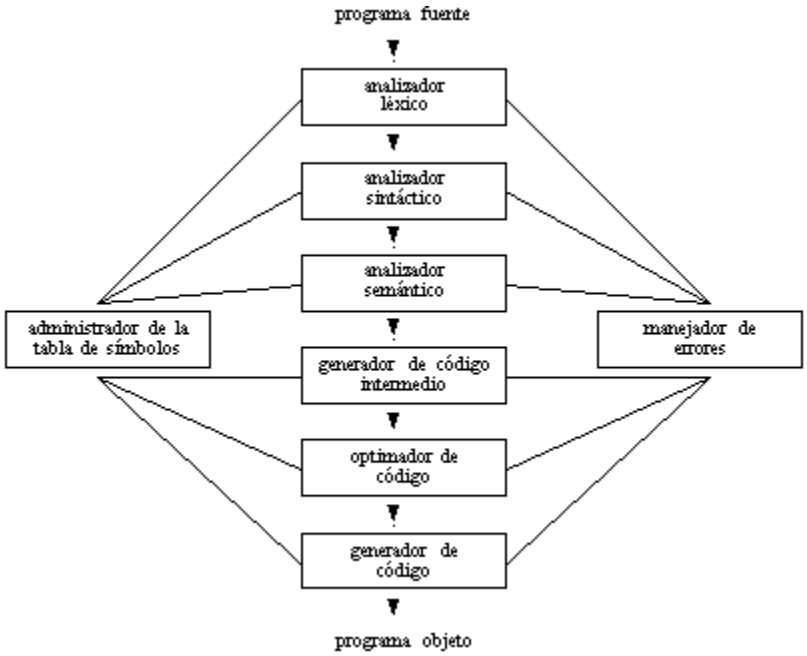

Figura 4.3 Fases de un compilador [Aho et. al 1995]

4.6 Interpretes

En lugar de producir un programa objeto como resultado de una traducción, un intérprete realiza las operaciones que implica el programa fuente. Por ejemplo un interprete podría construir un árbol como el de la figura 4.1 , y después efectuar las operaciones de los nodos conforme recorre el árbol.

Muchas veces los interpretes se utilizan para ejecutar lenguajes de órdenes, pues cada operador que se ejecuta en un lenguaje de este tipo suele ser una invocación de una rutina, como un editor o un compilador. Del mismo modo algunos lenguajes de alto nivel son interpretados, porque hay muchas cosas sobre los datos, como el tamaño y la forma de las matrices que no se pueden deducir en el momento de la compilación.

4.7 Lenguaje de Consultas Multibase de Datos

La división entre el análisis léxico y el análisis sintáctico es algo arbitraria. Generalmente se elige una división que simplifique la tarea completa del análisis. Un factor para determinar la división es si una construcción es inherentemente recursiva o no. Las construcciones léxicas no requieren recursión, mientras que las construcciones sintácticas suelen requerirla. Las gramáticas independientes de contexto son una formalización de reglas recursivas que se pueden usar para guiar el análisis sintáctico [Aho

et al. 1990].

4.7.1.3 Análisis Semántico

La fase de análisis semántico utiliza la estructura sintáctica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones. En el caso de un interprete de consultas se debe validar que los nombres de atributos y de relaciones sean válidos y tengan sentido desde el punto de vista semántico.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el interprete verifica si cada operador tiene operandos permitidos por la especificación del lenguaje fuente y verifica la compatibilidad de los tipos cuando estos están involucrados, por ejemplo, en una condición..

4.7.2 Detección e Información de Errores

En cada fase se pueden encontrar errores. Después de detectar un error cada fase debe informar del error con la descripción que le permita al usuario o programador ubicarlo y corregirlo. Los errores se dan con mayor frecuencia en las fases de análisis sintáctico y semántico. La fase léxica puede detectar errores donde los caracteres restantes de la entrada no forman ningún componente léxico del lenguaje. Los errores donde la cadena componentes léxicos violan las reglas de estructura del lenguaje son detectados por la fase de análisis sintáctico. En el análisis semántico el interprete detecta construcciones que tienen la estructura sintáctica correcta, pero que no tienen significado para la operación implicada, por ejemplo, si en un *select* se intenta seleccionar un atributo cuya tabla no se especifica en la cláusula *from*..

Posterior a la fase de análisis en un lenguaje de consultas se lleva a cabo la descomposición de la consulta global en subconsultas, después la recuperación de la información desde las diversas bases de datos y por ultimo la reconstrucción de la consulta global, mismas que ya se explicaron en el capítulo anterior.

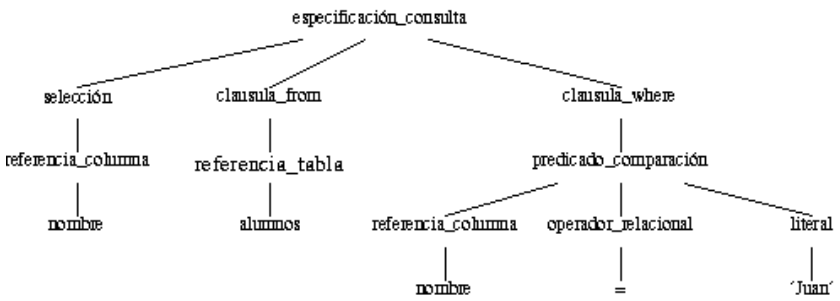

Figura 4.4 Árbol de análisis sintáctico
para: select nombre from alumnos where nombre=´Juan´;

4.8 Herramientas para la Construcción de Compiladores e Interpretes

Se han creado herramientas generales para el diseño automático de componentes específicos de compilador. Estas herramientas utilizan lenguajes especializados para especificar e implantar el componente, y pueden utilizar algoritmos bastante complejos. Las herramientas mas efectivas son las que ocultan los detalles del algoritmo y producen componentes que se pueden integrar con facilidad al resto del compilador.

4.8.1. Generadores de Analizadores Léxicos

Estos generadores generan automáticamente analizadores léxicos, por lo general a partir de una especificación basada en expresiones regulares [Aho et al. 1990].

Una herramienta muy utilizada para generar analizadores léxicos es LEX, esta ayuda al tomar un conjunto de descripciones de posibles componentes léxicos, y produce una rutina en C la cual se conoce como analizador léxico. Véase [Levine et al. 1995].

4.8.2. Generadores de Analizadores Sintácticos

Estos generadores producen analizadores sintácticos, normalmente a partir de una entrada fundamentada en una gramática independiente de contexto [Aho et al. 1990]. Muchos de estos generadores utilizan poderosos algoritmos de análisis sintáctico, y son demasiado complejos para realizarlos manualmente .

Uno de los generadores de analizadores sintácticos mas utilizados es YACC, el cual genera un analizador sintáctico tipo LALR [Aho et al. 1990]. YACC toma una descripción concisa de una gramática y produce rutinas en C que pueden analizar esta gramática. Un analizador YACC normalmente no es mas rápido que uno generado manualmente, pero es mas fácil de escribir y modificar. Información referente a YACC lo encuentra en [Levine et al. 1995].

4.8.3 JavaCC

Javacc es una herramienta escrita en java que genera analizadores

sintácticos con los analizadores léxicos incluidos. El analizador sintáctico generado incluye una declaración de un método publico por cada no-terminal en el archivo de la gramática. A diferencia de YACC no hay un solo símbolo inicial no terminal en JavaCC, uno puede iniciar el análisis sintáctico con respecto a cualquier no terminal en la gramática [Sun 1998].

La producción BNF (forma Backus-Naur) [Aho et al. 1990], es la producción estándar utilizada en la especificación de gramáticas en JavaCC. Cada producción BNF tiene un lado izquierdo, el cual es la especificación de un no terminal, la producción BNF después define este no terminal en términos de expansiones BNF en el lado derecho de la producción. En JavaCC los no-terminales son escritos exactamente como un método es declarado en java. El nombre del no-terminal es el nombre del método, y los parámetros y valores de retorno declarados son lo mismo que pasar por arriba y abajo del árbol de análisis sintáctico.

Javacc genera analizadores sintácticos tipo LL(k) [Aho et al. 1990]. Los analizadores sintácticos generados por Javacc toman decisiones en puntos de selección basados en alguna exploración de componentes léxicos por adelantado en la cadena de entrada y entonces deciden que camino seguir, es decir ningún retroceso es ejecutado, solo una decisión es hecha. En un analizador sintáctico LL(k), la *k* representa los componentes léxicos que se deben tomar por adelantado para llevar a cabo el análisis, y en Javacc la *k* puede ser tan grande como sea necesario.

4.8.3.1 JJTree de JavaCC

JJTree es un preprocesador para JavaCC que inserta arboles de análisis sintáctico (AAS) integrando acciones en varios lugares en la fuente JavaCC. La salida de JJTree es ejecutada a través de JavaCC para crear el analizador sintáctico.

Por defecto JJTree genera código para construir los nodos del AAS para cada no-terminal en el lenguaje . Este comportamiento puede ser modificado de manera que algunos no-terminales no tengan nodos generados.

JJTree define una interface java Node que todos los nodos del AAS deben implementar. La interface provee métodos para operaciones tales como: configurar el padre del nodo y agregar hijos y recuperarlos, además de que a cada nodo se le pueden asociar un conjunto de atributos como un tipo, una cadena, una posición de memoria o cualquier cosa. De esta manera con JJTree se puede manipular el árbol como se desee, lo cual facilita llevar a cabo algunas tareas tales como una *definición dirigida por la sintaxis* [Aho et al. 1990] .

En el apéndice Y se muestra la implementación del lenguaje propuesto en

este trabajo haciendo uso de JJtree y JavaCC.

En el presente capítulo se describieron la compilación y la interpretación como tecnologías para el desarrollo de lenguajes. Aunque como se puede observar nos inclinamos por describir a más detalle a los interpretes. Esto porque el prototipo para el lenguaje de consulta que se implemento es interpretado, se hizo esta elección debido a que el lenguaje de consulta solo procesa una consulta a la vez, por lo cual no resultaría ventajoso crear un compilador y también debido a la naturaleza del lenguaje Java el cual es un interprete .

Haciendo uso de JavaCC y Java en el siguiente capítulo se describe la forma en que se implemento un lenguaje de consulta multibase de datos como parte de esta tesis.