PostgresKDD

Un Sistema para el Descubrimiento de Conocimiento en Bases de Datos Fuertemente Acoplado con el SGBD PostgreSQL

Manual de Referencia

Director:

Ricardo Timarán Pereira, Ph.D.

Grupo de desarrollo:

Mario Guerrero Diaz

Mario Fernando Díaz

Camilo Cerquera Erazo

Stivenson Armero K.

Universidad de Nariño
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas
Grupo de Investigación GRIAS
Linea KDD
San Juan de Pasto
Colombia

2006

Contenido

Resumen 5

1 Introducción 6

2 El Proceso de Descubrimiento de Conocimiento en Bases de Datos 9

2.1 Definición. 9

2.2 Etapas del Proceso de DCBD 9

2.2.1 Etapa de Selección 10

2.2.2 Etapa de Preprocesamiento/Limpieza 10

2.2.3 Etapa de transformación/Reducción 11

2.2.4 Etapa de Minería de Datos 11

2.2.5 Etapa de Interpretación/Evaluación de Datos 11

2.3 Tareas de Minería de Datos implementadas en PostgresKDD 12

2.3.1 Tarea de Asociación 12

2.3.2 Tarea de Clasificación 13

3 Operadores Algebraicos para Asociación y Clasificación 13

3.1 Operadores Algebraicos para Asociación 13

3.1.1 Operador Associator (α) 13

3.1.2 Operador EquiKeep (χ) 14

3.1.3 Operador Describe Associator (βα) 15

3.2 Operadores Algebraicos para Clasificación 16

3.2.1 Operador Mate (χ) 17

3.2.2 Operador agregado Entro () 17

3.2.3 Operador agregado Gain () 18

3.2.4 Operador Describe Classifier (βμ) 18

4 Primitivas SQL para Asociación y Clasificación 19

4.1 Primitivas SQL para la tarea de Asociación 19

4.1.1 Primitiva Associator Range 20

4.1.2 Primitiva EquiKeep On 21

4.1.3 Operador Unificado SQL Describe Association Rules 22

4.2 Primitivas SQL para la tarea de Clasificación 23

4.2.1 Primitiva Mate By 23

4.2.2 Función agregada SQL Entro () 24
4.2.3 Función agregada SQL Gain () 24
4.2.4 Operador Unificado SQL Describe Classification Rules 25
5 Aspectos de Implementación de PostgresKDD en un acoplamiento fuerte con el SGBD PostgreSQL 26
5.1 Operadores en PostgreSQL 26
5.1.1 Un operador sencillo. 26
5.1.2 Un operador Complejo 28
5.1.3 Estructura general de un operador de PostgreSQL 29
5.2 Implementación de la primitiva Associator Range al interior del motor de PostgreSQL 33
5.2.1 Parser 33
5.2.2 Rewriter 37
5.2.3 Planner/Optimizar 37
5.2.4 Executor 40
5.3 Implementación de la primitiva EquiKepp On al interior del motor de PostgreSQL 43
5.3.1 Parser 43
5.3.2 Planner/Optimizar 46
5.3.3 Executor 50
5.4 Implementación de la primitiva Mate By al interior del motor de PostgreSQL 52
5.4.1 Parser 52
5.4.2 Planner/Optimizar 55
5.4.3 Executor 58
5.5 Implementación de los operadores Entro, Gain y Describe Classifier como funciones definidas por usuario (FDU) en PostgreSQL 60
5.5.1 Implementación de la FDU Entro() 60
5.5.2 Implementación de la FDU Gain() 63
5.6 Implementación del operador Describe Associator como la FDU Describe_Association_Rules en PostgreSQL 65
5.7 Implementación del operador Describe Classifier como la FDU Describe_Classification_Rules() en PostgreSQL 68
6 Proceso de Instalación de PostgresKDD 69
6.1 Requisitos del Sistema 69

6.2 Instalación y Configuración de PostgresKDD 70
6.3 Instalación y Configuración de las FDUs para Asociación y Clasificación 71
6.3.1 FDUs para Asociación 71
6.3.2 FDUs para Clasificación 72
Referencias Bibliográficas 74

Resumen

A pesar del acelerado avance y del incremento de la investigación en el área de Descubrimiento de Conocimiento en Bases de Datos, relativamente son pocas las propuestas de integrar al lenguaje de consultas SQL nuevas primitivas que permitan descubrir eficientemente conocimiento en grandes bases de datos. La mayor parte de los sistemas DCBD existentes se han construido bajo el modelo de arquitectura débilmente acoplada con un SGBD. La ventaja de esta arquitectura es su portabilidad. Sus principales desventajas son la escalabilidad y el rendimiento. Un Sistema DCBD fuertemente acoplado con un SGBD resuelve los problemas de escalabilidad y rendimiento de las otras arquitecturas, debido a que todos los algoritmos se integran al motor del SGBD como una primitiva y son ejecutados conjuntamente con los datos.

En este documento, se presenta la primera versión del proyecto de construir una herramienta computacional para el Descubrimiento de Conocimiento fuertemente acoplada con el SGBD PostgreSQL, denominada PostgresKDD. Aplicando el método tres-pasos se definieron e implementaron nuevos operadores algebraicos y primitivas SQL para el Descubrimiento de Reglas de Asociación y Clasificación al interior del motor del Sistema Gestor de Bases de Datos PostgreSQL. PostgresKDD fue desarrollada en el laboratorio de KDD del departamento de Ingeniería de Sistemas de la Universidad de Nariño (Colombia).

1. Introducción

El Descubrimiento de Conocimiento en Bases de Datos (DCBD) es el proceso no trivial de identificación de patrones válidos, novedosos, potencialmente útiles y fundamentalmente entendibles al usuario a partir de los datos (Fayyad et al., 1996). El proceso consiste en extraer patrones en forma de reglas o funciones, a partir de los datos, para que el usuario los analice. Esta tarea implica generalmente preprocesar los datos, hacer minería de datos (*Data Mining*) y presentar resultados.

Las investigaciones en DCBD, se centraron inicialmente en definir nuevas operaciones de descubrimiento de patrones y desarrollar algoritmos para éstas (Agrawal et al., 1993), (Agrawal y Srikant, 1994), (Park et al., 1995), (Brin et al., 1997). Investigaciones posteriores (Chaudhuri, 1998), (Agrawal y Shim, 1996), (Sarawagi et al., 2000), (Agrawal et al., 1996), (Han et al., 1996a), (Imielnski y Mannila, 1996), (Matheus et al., 1993), (Meo et al., 1996) se han enfocado en el problema de integrar DCBD con Sistemas Gestores de Bases de Datos (SGBD), haciendo de ésta un área activa de investigación. Los enfoques de integración de DCBD y SGBD reportados en la literatura se pueden ubicar en uno de tres tipos de arquitectura: sistemas débilmente acoplados, medianamente acoplados y fuertemente acoplados (Timarán, 2001).

Una arquitectura es débilmente acoplada cuando los algoritmos de Minería de Datos y demás componentes se encuentran en una capa externa al SGBD, por fuera del núcleo y su integración con éste se hace a partir de una interfaz-SQL (Timarán, 2001). Mientras el SGBD provee el almacenamiento persistente, la mayoría de procesamiento de datos se realiza en herramientas y aplicaciones por fuera del motor del SGBD (ver figura 1). Estos sistemas, a pesar de que son fáciles de integrar a cualquier SGBD, son ineficientes en términos de escalabilidad y rendimiento. El problema de escalabilidad consiste en que las herramientas y aplicaciones de este tipo de arquitectura, cargan todo el conjunto de datos en memoria, lo que las limita para el manejo de grandes cantidades de datos. El bajo rendimiento se debe a la copia de registros de la base de datos a la aplicación y al alto costo de las operaciones de entrada/salida cuando se manejan grandes volúmenes de datos.

Una arquitectura es medianamente acoplada cuando ciertas tareas y algoritmos de descubrimiento de patrones se encuentran formando parte del SGBD mediante procedimientos almacenados o funciones definidas por el usuario (FDU) (Timarán, 2001). La ventaja de este tipo de acoplamiento es que brinda mejor desempeño, ya que ejecutan todas las conexiones, aplicaciones y la base de datos en el mismo espacio de direccionamiento, además, un procedimiento almacenado o una FDU recibe igual trato que cualquier otro objeto de la base de datos y es registrado en el catálogo. La principal desventaja es el costo de desarrollo y la baja interoperabilidad ya que, para cada nueva instancia de una base de datos, se deben definir los procedimientos almacenados y las FDUs. En la figura 2 se muestra una arquitectura medianamente acoplada.

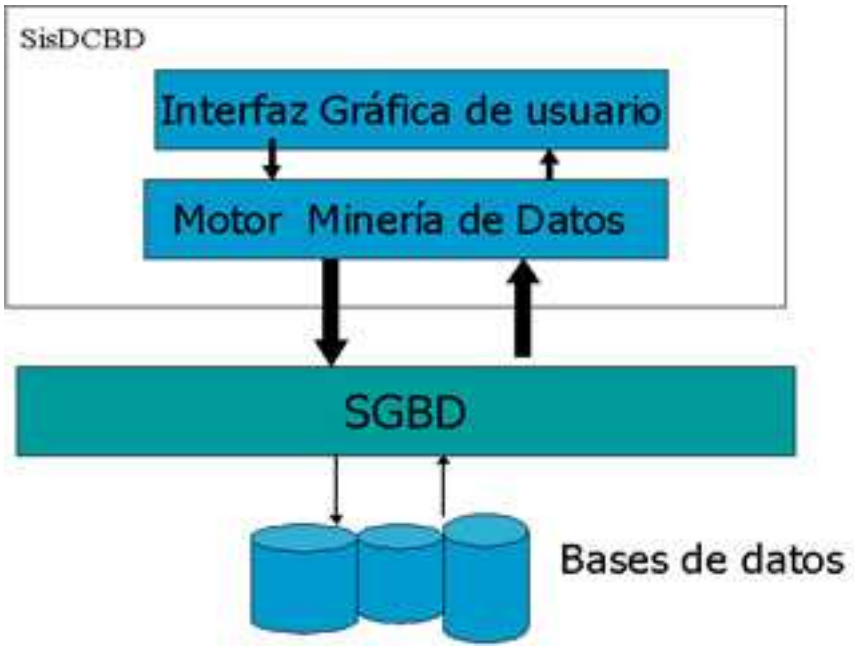

Figura 1. Arquitectura DCBD débilmente acoplada con un SGBD

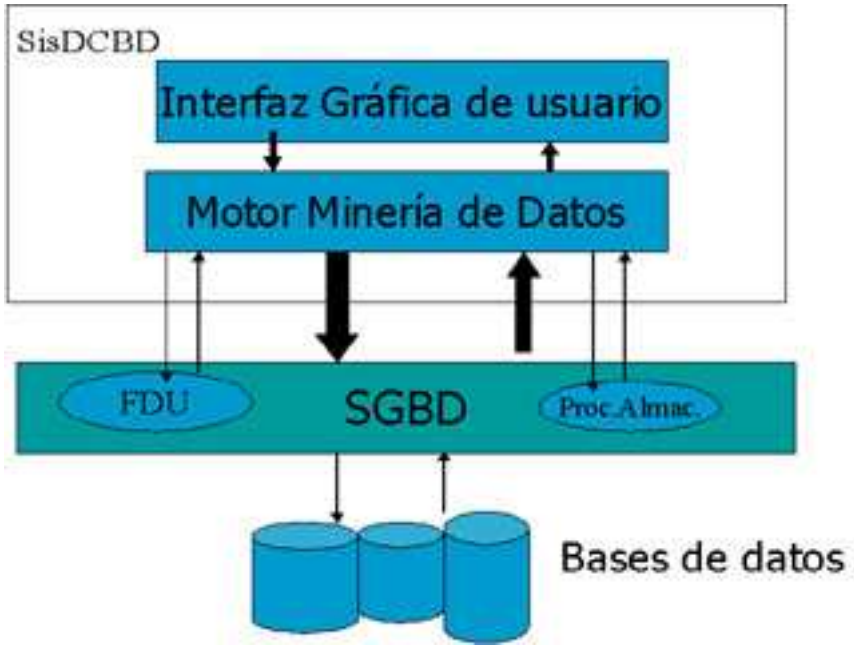

Figura 2. Arquitectura DCBD medianamente acoplada con un SGBD

Una arquitectura es fuertemente acoplada cuando la totalidad de las tareas y algoritmos de descubrimiento de patrones forman parte del SGBD como una operación primitiva, dotándolo de las capacidades de descubrimiento de conocimiento y posibilitándolo para desarrollar aplicaciones de este tipo (Timarán, 2001). Debido a que todos los algoritmos son ejecutados en el mismo espacio de direccionamiento que los datos en el SGBD, la ventaja potencial de este enfoque es que resuelve los problemas de escalabilidad y rendimiento de las otras arquitecturas (ver figura 3).

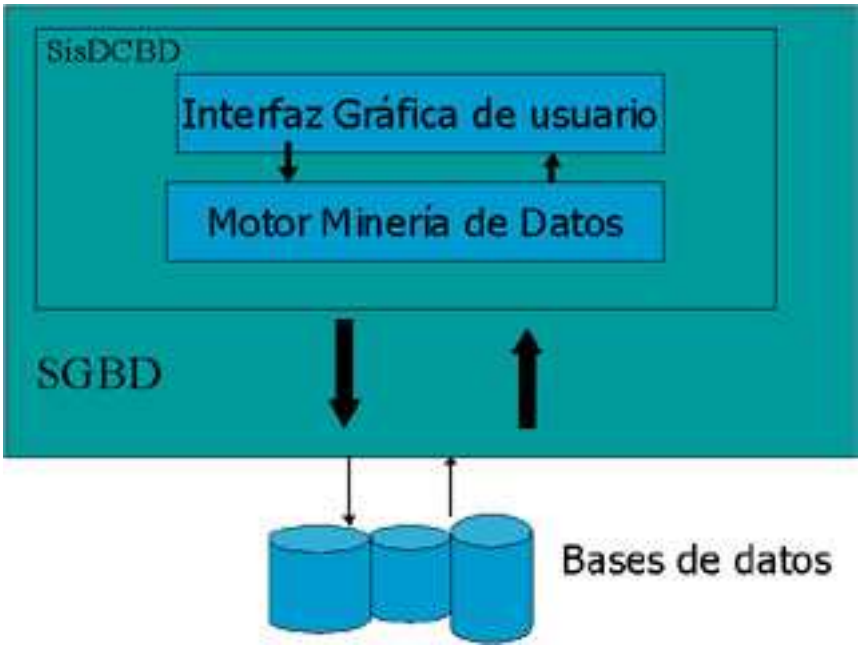

Figura 3. Arquitectura DCBD fuertemente acoplada con un SGBD

Hay propuestas de investigación que discuten la manera como tales sistemas pueden ser implementados: expresando ciertas operaciones de minería de datos como una serie de consultas SQL (Thomas y Sarawgi, 1998) (Wang et al., 1998) (Yoshizawa et al., 2000), diseñando e implementando nuevos lenguajes de consulta como extensiones del lenguaje SQL con nuevos operadores unificados, los cuales soportan ciertas tareas de minería de datos: DMQL (Han et al., 1996b), MSQL (Imielinski y Virmani, 1999), Mine Rule (Meo et al., 1998), OLE DB for Data Mining (Netz et al., 2000), SQL/MM (Melton y Eisenberg, 2001) (Schwenkreis, 2000), y definiendo nuevas primitivas SQL genéricas que facilitan el proceso de DCBD sin soportar una tarea en particular: NonStop SQL/Mx (Clear et al., 1999), Count by Group, Count by Order Group (Freitas y Lavington, 1997), FilterPartition, ComputeNodeStatics, PredictionJoin (Sattler y Dunemann, 2001). Otro enfoque es el método tres-pasos propuesto en (Timaran et al. 2003) que facilita la integración fuerte de una tarea de minería de datos con un SGBD. El método esta compuesto por tres pasos: en el primero, se extiende el álgebra relacional (Codd, 1970) con nuevos operadores algebraicos que faciliten los procesos computacionalmente más costosos de la tarea de minería de datos; en el segundo, se extiende el lenguaje SQL con nuevas primitivas expresadas en la cláusula SQL SELECT, las cuales implementan los nuevos operadores algebraicos y en el

tercero, se unifica estas primitivas en un nuevo operador SQL que se expresa en una nueva cláusula SQL y que obtiene el resultado de la tarea de minería de datos.

En este documento, se presenta la primera versión del proyecto de construir una herramienta computacional para el Descubrimiento de Conocimiento fuertemente acoplada con el SGBD PostgreSQL, denominada PostgresKDD. Aplicando el método tres-pasos (Timaran et al. 2003), se definieron e implementaron nuevos operadores algebraicos y primitivas SQL para el Descubrimiento de Reglas de Asociación y Clasificación al interior del motor del SGBD PostgreSQL. El resultado es un sistema DCBD fuertemente acoplado con un PostgreSQL con la capacidad de descubrir reglas de Asociación y Clasificación. PostgresKDD fue desarrollada en el laboratorio de KDD del departamento de Ingeniería de Sistemas de la Universidad de Nariño (Colombia).

El resto del documento se organiza de la siguiente manera: en la sección 2, se dan los conceptos preliminares sobre el proceso de descubrimiento de conocomiento en bases de datos y las tares de minería de datos asociación y clasificación. En la sección 3, se describen los operadores algebraicos con los cuales se extiende el álgebra relacional para soportar las tares de asociación y clasificación. En la sección 4, se presentan las primitivas SQL y operadores unificados, con los cuales se extiende el lenguaje SQL para descubrir reglas de Asociación y Clasificación. En la sección 5, se muestra la manera como se implementan las primitivas y operadores unificados SQL para Asociación y Clasificación al interior del SGBD PostgreSQL. En la sección 6 se describe el proceso de instalación de PostgresKDD y las funciones definidas por el usuario.

2. El Proceso de Descubrimiento de Conocimiento en Bases de Datos

2.1 Definición.

En la definición del proceso de descubrimiento de conocimiento en bases de datos, la mayoría de autores coinciden con (Fayyad et al., 1996) en definirlo como:

El proceso no trivial de identificación de patrones válidos, novedosos, potencialmente útiles y fundamentalmente entendibles al usuario a partir de los datos.

2.2 Etapas del Proceso de DCBD.

El proceso de DCBD, que se muestra en la figura 4, es interactivo e iterativo, involucra numerosos pasos con la intervención del usuario en la toma de muchas decisiones y se resumen en las siguientes etapas:

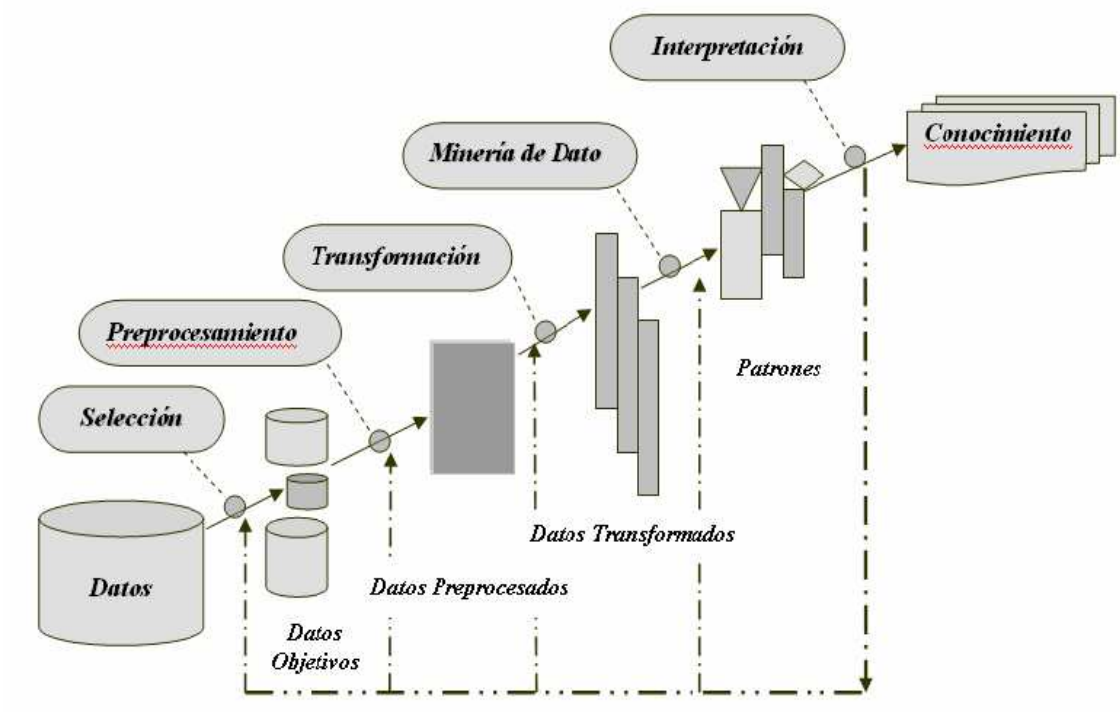

Figura 4. Etapas del proceso DCBD.

- Selección
- Preprocesamiento/Limpieza
- Transformación/Reducción
- Minería de Datos (Data Mining)
- Interpretación / evaluación

2.2.1 Etapa de Selección.

En la etapa de Selección, una vez identificado el conocimiento relevante y prioritario y definidas las metas del proceso DCBD, desde el punto de vista del usuario final, se crea un conjunto de datos objetivo, seleccionando todo el conjunto de datos o una muestra representativa de éste, sobre el cual se va ha realizar el proceso de descubrimiento. La selección de los datos varía de acuerdo con los objetivos del negocio.

2.2.2 Etapa de Preprocesamiento/Limpieza.

En la etapa de Preprocesamiento/Limpieza (*Data cleaning*) se analiza la calidad de los datos, se aplican operaciones básicas como la remoción de datos ruidosos, se seleccionan estrategias para el manejo de datos desconocidos (*missing* y *empty*), datos nulos, datos duplicados y técnicas estadísticas para su reemplazo. En esta etapa, es de suma importancia la interacción con el usuario o analista.

Los datos ruidosos (noisy data) son valores que están significativamente fuera del rango de valores esperado. Se deben principalmente, a errores humanos, a cambios en el sistema, a información no disponible a tiempo y a fuentes heterogéneas de datos. Los datos desconocidos empty son aquellos a los cuales no les corresponde un valor en el mundo real y los missing son aquellos que tienen un valor que no fue capturado. Los datos nulos son datos desconocidos que son permitidos por los sistemas gestores de bases de datos relacionales (SGBDR). En el proceso de limpieza todos estos valores se ignoran, se reemplazan por un valor por omisión, por el valor más cercano o se usan métricas de tipo estadístico como la media, mínimo, máximo, para reemplazarlos.

2.2.3 Etapa de transformación/Reducción.

En la etapa de transformación/reducción de datos, se buscan características útiles para representar los datos dependiendo de la meta del proceso. Se utilizan métodos de reducción de dimensiones o de transformación para disminuir el número efectivo de variables bajo consideración o para encontrar representaciones invariantes de los datos.

Los métodos de reducción de dimensiones pueden simplificar una tabla de una base de datos horizontalmente o verticalmente. La reducción horizontal implica la eliminación de tuplas idénticas como producto de la sustitución del valor de un atributo por otro de alto nivel, en una jerarquía definida de valores categóricos o por la discretización de valores continuos (e.g., edad por un rango de edades). La reducción vertical implica la eliminación de atributos que son insignificantes o redundantes con respecto al problema, como la eliminación de llaves, la eliminación de columnas que dependen funcionalmente (e.g., edad y fecha de nacimiento). Se utilizan técnicas de reducción tales como agregaciones, compresión de datos, histogramas, segmentación, discretización basada en entropía, muestreo entre otras (Han y Kamber 2001).

2.2.4 Etapa de Minería de Datos.

El objetivo de la etapa minería de datos es la búsqueda y descubrimiento de patrones insospechados y de interés aplicando tareas de descubrimiento tales como clasificación, *clustering*, patrones secuenciales y asociaciones, entre otras.

La escogencia de un algoritmo de minería de datos incluye: la selección de los métodos a aplicar en la búsqueda de patrones en los datos, así como la decisión sobre los modelos y los parámetros mas apropiados, dependiendo del tipo de datos(categóricos, numéricos) a utilizar.

2.2.5 Etapa de Interpretación/Evaluación de Datos.

En la etapa de interpretación/evaluación, se interpretan los patrones descubiertos y posiblemente se retorna a las anteriores etapas para posteriores iteraciones. Esta etapa, puede incluir la visualización de los patrones extraídos, la remoción de los patrones redundantes o irrelevantes y la traducción de los patrones útiles en términos que sean entendibles para el usuario. Por otra parte, se consolida el conocimiento descubierto para

incorporarlo en otro sistema para posteriores acciones, o simplemente para documentarlo y reportarlo a las partes interesadas, asi como también para verificar y resolver conflictos potenciales con el conocimiento previamente descubierto.

2.3 Tareas de Minería de Datos implementadas en PostgresKDD

2.3.1 Tarea de Asociación

El problema de encontrar reglas de asociación fue formulado por Agrawal et al. (Agrawal et al., 1993) (Agrawal y Srikan, 1994) y a menudo se referencia como el problema de canasta de mercado (*market-basket*). En este problema se da un conjunto de items y una colección de transacciones que son subconjuntos (canastas) de estos items. La tarea es encontrar relaciones entre la presencia de varios items en esas canastas.

Formalmente, sea $I= \{i_1, i_2, ...,i_m\}$ un conjunto de literales, llamados items. Sea D un conjunto de transacciones, donde cada transacción T es un conjunto de items tal que $T \subseteq I$. Cada transacción se asocia con un identificador, llamado *TID*. Sea X un conjunto de items. Se dice que una transacción T contiene a X si y solo si $X \subseteq T$. Una regla de asociación es una implicación de la forma $X \Rightarrow Y$, donde X y Y son conjuntos de items tal que $X \subset I$, $Y \subset I$ y $X \cap Y = \phi$. El significado intuitivo de tal regla es que las transacciones de la base de datos que contienen X tienden a contener Y. La regla $X \Rightarrow Y$ se cumple en el conjunto de transacciones D con una confianza c si el $c\%$ de las transacciones en D que contienen X también contienen Y. La regla $X \Rightarrow Y$ tiene un soporte s en el conjunto de transacciones D si el $s\%$ de las transacciones en D contienen $X \cup Y$.

Un ejemplo de una regla de asociación es: “el 30% de las transacciones que contienen cerveza también contienen pañales; el 2% de todas las transacciones contienen ambos items” [Agrawal et al., 1996]. Aquí el 30% es la confianza de la regla y el 2%, el soporte de la regla.

La confianza denota la fuerza de la implicación y el soporte indica la frecuencia de ocurrencia de los patrones en la regla. Las reglas con una confianza alta y soporte fuerte son referidas como reglas fuertes (*strong rules*) (Agrawal y Srikan, 1994). El problema de encontrar reglas de asociación se descompone en los siguientes pasos:

- Descubrir los itemsets frecuentes, i.e., el conjunto de itemsets que tienen el soporte de transacciones por encima de un predeterminado soporte *s* mínimo.

- Usar los itemsets frecuente para generar las reglas de asociación para la base de datos.

Después de que los itemsets frecuentes son identificados, las correspondientes reglas de asociación se pueden derivar de una manera directa.

2.3.2 Tarea de Clasificación

La clasificación de datos es el proceso por medio del cual se encuentra propiedades comunes entre un conjunto de objetos de una base de datos y se los clasifica en diferentes clases, de acuerdo al modelo de clasificación (Han y Kamber, 2001). Este proceso se realiza en dos pasos: en el primer paso se construye un modelo en el cual, cada tupla, de un conjunto de tuplas de la base de datos, tiene una clase conocida (etiqueta), determinada por uno de los atributos de la base de datos, llamado *atributo clase*. El conjunto de tuplas que sirve para construir el modelo se denomina *conjunto de entrenamiento*. Cada tupla de este conjunto es un ejemplo de entrenamiento. En el segundo paso, se usa el modelo para clasificar. Inicialmente, se estima la exactitud del modelo utilizando un conjunto de tuplas de la base de datos, generalmente diferente al de entrenamiento, cuya clase es conocida, denominado *conjunto de prueba*. A cada tupla de este conjunto se denomina ejemplo de prueba.

La exactitud del modelo, sobre el conjunto de prueba, es el porcentaje de ejemplos de prueba que son correctamente clasificadas por el modelo. Si la exactitud del modelo se considera aceptable, se puede usar para clasificar futuros datos o tuplas para los cuales no se conoce la clase a la cual pertenecen.

3. Operadores Algebraicos para Asociación y Clasificación

3.1 Operadores Algebraicos para Asociación.

Para garantizar la eficiencia en las operaciones de minería de datos, un nuevo operador algebraico debe facilitar los procesos de minería de datos computacionalmente más costosos. En asociación, el cálculo de los *itemsets* frecuentes, i.e. todos aquellos *itemsets* cuyo soporte es mayor o igual a un soporte mínimo, determina el rendimiento total del proceso de encontrar reglas de asociación (Chen et al., 1996). Un nuevo operador algebraico para asociación debe facilitar este proceso.

Los operadores algebraicos unarios propuestos por Timarán (Timarán et al, 2003) (Timarán y Millan, 2005a) (Timarán y Millan, 2005b) y con los cuales se extiende el álgebra relacional para facilitar el soportar la tarea de Asociación son *Associator*, *Equikeep* y *Describe Associator*.

3.1.1 Operador *Associator* (α)

Associator(α) es un operador algebraico unario que genera, a partir de cada tupla de una relación, todas las posibles combinaciones (de diferente tamaño) de los valores de sus atributos, como tuplas de una nueva relación, en una sola pasada. Este operador conserva el esquema de la relación inicial y hace nulos todos aquellos valores de los atributos de la tupla, que no forman la combinación. Los atributos de la relación inicial pueden pertenecer a diferentes dominios. Su sintaxis es:

$$\alpha_{\text{tamaño\_inicial, tamaño\_final}}(R)$$

donde <*tamano_inicial* > y <*tamaño-final*> son dos parámetros de entrada que determinan el tamaño inicial y tamaño final de los *itemsets* . *R* es la relación inicial.

Ejemplo 1. Sea la relación *R*(*A*,*B*,*C*,*D*) de la figura 5. Encontrar las diferentes combinaciones de tamaño 2 hasta tamaño 4, es decir $R1=\alpha_{2,4}(R)$.

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2

Figura 5. Relación *R*

El resultado del operador Associator aplicado al ejemplo de la Figura 5 se muestra en la figura 6.

A	B	C	D
a1	b1	null	null
a1	null	c1	null
a1	null	null	d1
null	b1	c1	null
null	b1	null	d1
null	null	c1	d1
a1	b1	c1	null
a1	b1	null	d1
a1	null	c1	d1
null	b1	c1	d1
a1	b1	c1	d1
a1	b2	null	null
a1	null	c1	null
a1	null	null	d2
null	b2	c1	null
null	b2	null	d2
null	null	c1	d2
a1	b2	c1	null
a1	b2	null	d2
a1	null	c1	d2
null	b2	c1	d2
a1	b2	c1	d2

Figura 6. Resultado operación $R1=\alpha_{2,4}(R)$

3.1.2 Operador *EquiKeep* (χ)

EquiKeep (χ) es un operador unario, que se asemeja a la *Restricción* (σ) por tener una expresión lógica que evaluar sobre una relación *R*, y conserva su esquema. Se diferencia de la *Restricción* en que en lugar de aplicar la condición a las filas (tuplas) de la relación, *EquiKeep* aplica la expresión lógica a las columnas (atributos) de *R*, es decir restringe los valores de los atributos de cada una de las tuplas de la relación *R*, a únicamente aquellos

que satisfacen una condición determinada, haciendo nulos al resto de valores y conservando el esquema de la relación. Su sintaxis es la siguiente:

$$\chi_{\text{expresión\_lógica}}(R)$$

donde *<expresión lógica>* es la condición que deben cumplir los valores de los atributos de la relación R para no hacerse nulos.

Ejemplo 2. Sea la relación R(A,B,C,D) de la figura 7. Restringir los valores de los atributos A=a1, B=b1 , C=c2 y D=d1, es decir, $R1=\chi_{A=a1\ v\ B=b1\ v\ C=c2\ v\ D=d1}(R)$. El resultado del operador *EquiKeep* se muestra en la figura 8.

A	B	C	D
a1	B1	c1	d1
a1	B2	c1	d2
a2	B2	c2	d2
a2	B1	c1	d1
a2	B2	c1	d2
a1	B2	c2	d1

Figura 7. Relación *R*

A	B	C	D
a1	b1	null	d1
a1	Null	null	null
null	Null	c2	null
null	b1	null	d1
null	Null	null	null
a1	Null	c2	d1

Figura 8. Resultado operación $R1=\chi_{A=a1\ v\ B=b1\ v\ C=c2\ v\ D=d1}$ (R)

En este ejemplo, la tupla {*a2,b2,c1,d2*} es eliminada por resultar todos sus valores nulos.

3.1.3 Operador *Describe Associator* (βα)

Describe Associator(βα) es un operador unario que toma como entrada la relación resultante del operador *associator* y por cada tupla de esta relación, genera, a partir de los atributos *l* no nulos de la tupla, todos los diferentes subconjuntos de un tamaño específico de la forma {{a}, {l-a},s}, donde {a} se denomina subconjunto antecedente y {l-a} subconjunto consecuente. {a} y {l-a}, son subconjuntos de atributos de *l* y *s* es el tamaño del subconjunto antecedente {a}.La sintaxis del operador *Describe Associator* es:

$$\beta\alpha_{\text{longitud\_regla}}(R)$$

donde *<longitud_regla>* es la longitud máxima de atributos no nulos por tupla.

Describe Associator facilita la generación de reglas de asociación tanto unidimensionales como multidimensionales (Han y Kamber, 2001).

Ejemplo 3. Sea la relación R(A,B,C) de la figura 9. Obtener los diferentes subconjuntos de tamaño 3 con el operador *Describe Associator,* es decir R1= $\beta\alpha_3(R)$. El resultado se muestra en la figura 10.

A	B	C
a1	b1	c1
a2	b2	c2

Figura 9. Relación R

A1	A2	A3	S
a1	b1	c1	1
b1	a1	c1	1
c1	a1	b1	1
a1	b1	c1	2
a1	c1	b1	2
b1	c1	a1	2
a2	b2	c2	1
b2	a2	c2	1
c2	a2	b2	1
a2	b2	c2	2
a2	c2	b2	2
c2	b2	a2	2

Figura 10. Relación R1=$\beta\alpha_3$(R).

3.2 Operadores Algebraicos para Clasificación.

La clasificación por árboles de decisión es, probablemente, el modelo más utilizado y popular por su simplicidad y facilidad para su entendimiento (Han y Kamber, 2001) (Sattler y Dunemann, 2001). Se han propuesto varias métricas para este proceso. El cálculo del valor de la métrica que permite seleccionar, en cada nodo, el atributo que tenga una mayor potencia para clasificar sobre el conjunto de valores del atributo clase, es la parte más costosa del algoritmo utilizado (Wang et al., 1998). Los algoritmos *ID3* (Quinlan, 1986) y *C4.5* (Quinlan, 1993) utilizan como métrica, para seleccionar el atributo candidato en cada nodo del árbol, la reducción de la entropía denominada *Ganancia de Información*. Para el cálculo de estas métricas, no se necesitan los datos en si, sino las estadísticas acerca del número de registros en los cuales se combinan los atributos condición con el atributo clase. Un nuevo operador algebraico para clasificación basado en árboles de decisión debe facilitar estas combinaciones, que conjuntamente con operadores agregados, permita el cálculo de estas métricas.

Los operadores algebraicos propuestos por Timarán (Timarán y Millán, 2006) (Timarán, 2007) y con los cuales se extiende el álgebra relacional para soportar la tarea de Clasificación son *Mate, Entro, Gain* y *Describe Classifier*.

3.2.1 Operador *Mate* (μ)

El operador *Mate* genera, por cada una de las tuplas de una relación, todas los posibles combinaciones formadas por los valores no nulos de los atributos pertenecientes a una lista de atributos denominados *Atributos Condición*, y el valor no nulo del atributo denominado *Atributo Clase*. El operador *Mate* genera estas combinaciones, en una sola pasada sobre la tabla de entrenamiento (lo que redunda en la eficiencia del proceso de construcción del árbol de decisión). *Mate* tiene la siguiente sintaxis:

$$\mu_{\text{lista\_atributos\_condición; atributo\_clase}}(R)$$

donde <*lista_atributos_condición*> es el conjunto de atributos de la relación *R* a combinar con el atributo clase y <*atributo_clase*> es el atributo de *R* definido como clase.

Ejemplo 4. Sea la relación *R*(*A,B,C,D*) de la figura 11 obtener las diferentes combinaciones de los atributos *A,B* con el atributo *D*, es decir, R1= $\mu_{A,B;D}(R)$.

El resultado de la operación R1= $\mu_{A,B;D}(R)$, se muestra en la figura 12.

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2

Figura 11. Relación *R*

A	B	D
a1	null	d1
null	b1	d1
a1	b1	d1
a1	null	d2
null	b2	d2
a1	b2	d2

Figura 12. Resultado operación R1=$\mu_{A,B;D}(R)$

3.2.2 Operador agregado Entro

El operador agregado *Entro* permite calcular la entropía de una relación R con respecto a un atributo denominado atributo condición y un atributo clase.

Tiene la siguiente sintaxis:

Entro(*atributo; atributo_clase;R*)

donde <*atributo*> es el atributo condición de la relación *R* y <*atributo_clase*> es el atributo con el que se combina el atributo <*atributo*>.

La entropía de *R* con respecto al atributo condición A_k es:

$$Entro(A_k;Ac;\ R)=\{y \mid y = -\sum p_{ij} \log_2(p_{ij}),\ \ 1 \leq i \leq t\ ,\ 1 \leq j \leq q,\ \ \ p_{ij} = s_{ij} / \mid S_j \mid \}$$

donde

- $p_{ij} = s_{ij} / \mid S_j \mid$ es la probabilidad que una tupla en S_j pertenezca a la clase C_i.

La entropía de *R* con respecto al atributo clase *Ac* es:

$$Entro(Ac;Ac;\ R)=\{y \mid y = -\sum p_i \log_2(p_i),\ \ 1 \leq i \leq t,\ \ p_i = \ r_i / m\}$$

donde:

- p_i es la probabilidad que un tupla cualquier pertenezca a la clase C_i
- r_i el número de tuplas de *r(A)* que pertenecen a la clase C_i.

3.2.3 Operador agregado Gain

El operador agregado *Gain* permite calcular la reducción de la entropía causada por el conocimiento del valor de un atributo de una relación.

Su sintaxis es:

Gain(*atributo*;*atrib_clase*;*R*)

donde <*atributo*> es el atributo condición de la relación *R* y <*atributo_clase*> es el atributo con el que se combina el atributo <*atributo*>.

Gain() permite calcular la ganancia de información obtenida por el particionamiento de la relación *R* de acuerdo con el atributo <*atributo*> y y se define:

$$Gain(A_k;Ac;\ R)=\{y \mid y = Entro(Ac;Ac;\ R) - Entro(A_k;Ac;\ R)\}$$

donde:

- *Entro* (*Ac; Ac; R*) es la entropía de la relación *R* con respecto al atributo clase *Ac*
- $Entro(A_k;Ac;\ R)$ es la entropía de la relación *R* con respecto al atributo A_k .

3.2.4 Operador Describe Classifier (βμ).

Describe Classifier (βμ) es un operador unario que toma como entrada la relación resultante de los operadores *Mate*, *Entro* y *Gain* y produce una nueva relación donde se almacenan los valores de los atributos que formarán los diferentes nodos del arbol de decisión.

La sintaxis del operador *Describe Classifier* es la siguiente:

βμ(R)

Formalmente, sea $A=\{A_1, .., A_n, E, G\}$ el conjunto de atributos de la relación R de grado $n+2$ y cardinalidad m. El operador $\beta\mu$ aplicado a R:

$$\beta\mu\ (R) = \{\ t_i(Y) \mid\ Y=\{N,P,A,V,C\ \}$$

donde:

- t_i=<*val(N),null,val(A),null,null*> si t_i es nodo raiz,
- t_i=<*val(N),val(P),val(A),val(V),val(C)*> si t_i es nodo hoja
- t_i=<*val(N),val(P),val(A),val(V),null*> si t_i es nodo interno

produce una nueva relación con esquema *R(Y), Y={N,P,A,V,C }* donde *N* es el número del nodo, *P* identifica al nodo padre, *A* identifica el nombre del atributo asociado a ese nodo, V es un valor para el atributo *A* y *C* es el atributo clase. Su extensión *r (Y),* está formada por un conjunto de tuplas en las cuales si los valores de los atributos son: N≠null, P=null, A≠null,V=null y C=null corresponde a un nodo raiz; si N≠null, P≠null, A≠null,V≠null y C≠null corresponde a una hoja o nodo terminal y si N≠null, P≠null, A≠null,V≠null y C=null corresponde a un nodo interno.

Describe Classifier facilita la construcción del árbol de decisión y por consiguiente la generación de reglas de clasificación.

Ejemplo 5. Sea la relación ARBOL(NODO,PADRE,ATRIBUTO,VALOR,CLASE) de la figura 13 resultado del operador *Describe Classifier* . Construir el árbol de decisión. El resultado se muestra en la figura 14.

NODO	PADRE	ATRIBUTO	VALOR	CLASE
N0	Null	Temperatura	Null	Null
N1	N0	Temperatura	Alta	Si
N2	N0	Temperatura	Media	No
N3	N0	Temperatura	Normal	Null
N4	N3	D_muscular	Si	Si
N5	N3	D_muscular	No	No

Figura 13. Relación Árbol

4. Primitivas SQL para Asociación y Clasificación

4.1 Primitivas SQL para la tarea de Asociación

Los operadores algebraicos *Associator* y *EquiKeep* extienden el álgebra relacional para soportar la tarea de minería de datos Asociación. Con el fin de que estos operadores se puedan expresar en el lenguaje SQL, se implementan como primitivas SQL dentro de la cláusula *SELECT*. El operador algebraico *Describe Associator* se implementa en el lenguaje SQL como un operador unificado en una cláusula independiente. De esta forma,

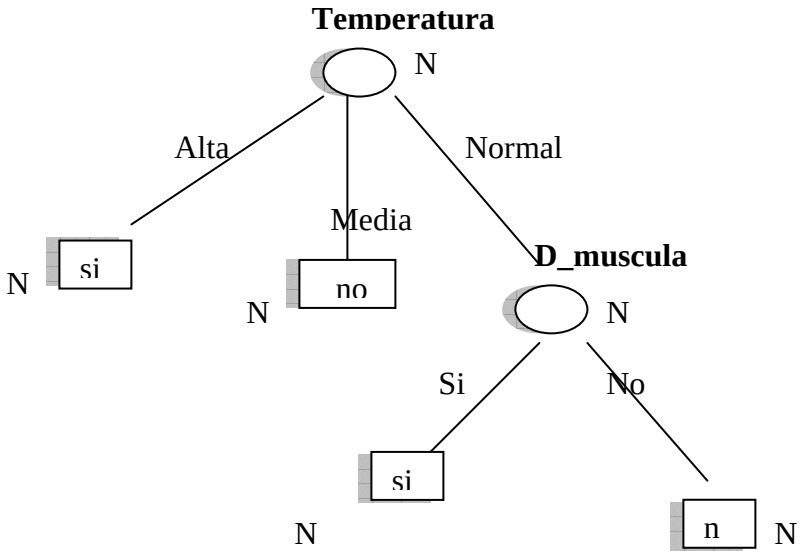

Figura 14. Árbol de decisión

el lenguaje SQL es capaz de soportar el descubrimiento de reglas de Asociación (Timarán et al, 2003) (Timarán y Millán, 2005b).

4.1.1 Primitiva *Associator Range*

Esta primitiva implementa el operador algebraico *Associator* en la cláusula SQL SELECT. *Associator Range* permite obtener por cada tupla de una tabla, todos los posibles subconjuntos desde un tamaño inicial hasta un tamaño final determinado por la cláusula RANGE.

Dentro de la cláusula SELECT, la primitiva *Associator Range* tiene la siguiente síntaxis:

SELECT <*ListaAtributosTablaDatos*> [INTO <*NombreTablaAssociator*>]
FROM <*NombreTablaDatos*>
WHERE <*CláusulaWhere*>
ASSOCIATOR RANGE <*valor1*> UNTIL <*valor2*>
GROUP BY <*ListaAtributosTablaDatos*>

<*ListaAtributosTablaDatos*> : := <*Atributo*>, <*ListaAtributos*>
<*ListaAtributos*> : := <*Atributo*>, <*ListaAtributos*>
<*ListaAtributos*> : := <*Atributo*>
<*valor1*> : := *1,2,3, 4 ...*
<*valor2*> : := *1,2,3,4 ...*

La primitiva *Associator Range* facilita el cálculo de los *itemsets* frecuentes para el descubrimiento de Reglas de Asociación en tablas multicolumna (Rajamani et al., 1999).

Ejemplo 6. Obtener de la tabla ESTUDIANTES (PROGRAMA, EDAD, SEXO, ESTRATO, PROMEDIO), los *itemsets* frecuentes de tamaño 2 y 3 que cumplan con un soporte mínimo mayor o igual a 2.

Los *itemsets* frecuentes se obtienen con la siguiente sentencia SQL:

```
SELECT programa,sexo, estrato, count(*) AS soporte INTO assostudent
FROM estudiantes
ASSOCIATOR RANGE 2 UNTIL 3
GROUP BY programa,sexo,estrato HAVING count(*)>=2
```

4.1.2 Primitiva *EquiKeep On*

Esta primitiva implementa el operador algebraico *EquiKeep* en la cláusula SQL SELECT. *EquiKeep On* conserva en cada registro de una tabla los valores de los atributos que cumplen una condición determinada. El resto de valores de los atributos se hacen nulos.

Dentro de la cláusula SELECT, *EquiKeep On* tiene la siguiente síntaxis:

```
SELECT <ListaAtributosTablaDatos> [INTO <NombreTablaEquiKeep>]
FROM  <NombreTablaDatos>
WHERE <CláusulaWhere>
EQUIKEEP ON  <CondiciónValoresAtributos >
GROUP BY <ListaAtributosTablaDatos>

<ListaAtributosTablaDatos> : := <Atributo>, <ListaAtributos>
<ListaAtributos> : := <Atributo>, <ListaAtributos>
<ListaAtributos> : := <Atributo>
<CondiciónValoresAtributos>::=<Atributo><operador><Valor>, <ListaCondición>  |
                                <Atributo><operador><(Listavalores)>  ,
                                <ListaCondición>

<ListaCondición>::=<Atributo=Valor>|<Atributo><operador><( Listavalores)>
<operador> : := AND, OR,NOT,IN
<Listavalores> : := 1,2,3, 4 ...
<Valor> : := 1,2,3, 4 ...
```

La primitiva EQUIKEEP ON facilita la generación de los *itemsets* frecuentes en el descubrimiento de Reglas de Asociación, al permitir conservar en cada registro de una tabla únicamente los valores de los atributos frecuentes o *itemsets* frecuentes. Se puede utilizar con los algoritmos *Apriori* (Agrawal y Srikant, 1994), *FP-Tree* (Han et al., 2000) o conjuntamente con la primitiva de Asociación *Associator*.

Ejemplo 7. Sea la tabla TRANSACCION (TID, ID_ITEM1, ID_ITEM2, ID_ITEM3). Encontrar los *itemsets* frecuentes de tamaño 2 y 3 formados por lo atributos ID_ITEM1, ID_ITEM2, ID_ITEM3 que cumplan un soporte mínimo mayor o igual a 3. Almacenar los resultados en la tabla EQUITRAN.

Suponiendo que ya se conocen los *itemsets* frecuentes de tamaño 1: {i2:4}, {i3:4}, {i4:3}, la sentencia SQL que permite obtener esta consulta utilizando las primitivas *EquiKeep On* y *Associator Range* es la siguiente:

```
SELECT  id_item1, id_item2,  id_item3, count(*) AS soporte  INTO equitran
FROM transaccion
EQUIKEEP ON id_item1 IN (i2,i3,i4) ,  id_item2 IN (i2,i3,i4) ,  id_item3 IN (i2,i3,i4)
ASSOCIATOR  RANGE  2 UNTIL 3
GROUP BY id_item1, id_item2,  id_item3 HAVING count(*)>=3
```

4.1.3 Operador Unificado SQL Describe Association Rules.

El operador SQL *Describe Association Rules* implementa el operador algebraico *Describe Associator* en una nueva cláusla SQL. *Describe Association Rules* permite generar, una vez calculados los *itemsets* frecuentes, todas las reglas de asociación unidimensionales o multidimensionales (Han y Kamber, 2001) de una longitud específica, que cumplen con un soporte y una confianza mayor o igual que unos determinados umbrales especificados por el usuario.

Describe Association Rules tiene la siguiente sintaxis:

```
DESCRIBE [UNIDIMENSIONAL][MULTIDIMENSIONAL] ASSOCIATION RULES
[INTO <TablaReglasAsociación>]
FROM  <NombreTablaItemsetsFrecuentes>
WITH CONFIDENCE  <valor1>
LENGTH <valor2 >
[DO   <SubconsultaItemsetsFrecuentes>]

<valor 1> ::=1,2,3,4,….
<valor 2> ::=1,2,3,4,….
< SubconsultaItemsetsFrecuentes >::=<SFWEAG>|<SFWEARG>|<SFWEACG>
<SFWEAG>   ::= <SELECT FROM WHERE EQUIKEEP ASSOCIATOR GROUP BY>
<SFWEARG>::= <SELECT FROM WHERE EQUIKEEP ASSOROW GROUP BY>
<SFWACG>::= <SELECT FROM WHERE   ASSOCOL GROUP BY>
```

Ejemplo 8. Sea la tabla CLIENTES (EDAD, OCUPACION,COMPRA) cuyos valores han sido discretizados. Encontrar las reglas de asociación multidimensionales de longitud 3, con un soporte minimo de 2 y una confianza mínima de 30. Las reglas generadas almacenarlas en la tabla ReglasAsociacion. Se supone que los *itemsets* frecuentes de tamaño 1 son *edad* IN (20...29,30...40), *ocupación* IN (estudiante, ingeniero, medico) y *compra* IN (televisor, laptop, impresora).

La sentencia SQL unificada que genera las reglas de asociación multidimensionales es:

```
DESCRIBE MULTIDIMENSIONAL ASSOCIATION RULES
INTO ReglasAsociacion
FROM  ItemsFrecuentes
WITH  CONFIDENCE  30
LENGTH 3
DO      SELECT  edad, ocupación,compra, count(*) AS soporte  INTO ItemsFrecuentes
```

```
FROM clientes
EQUIKEEP ON edad IN (20..29,30..40) ,
                ocupación IN (estudiante, ingeniero, medico) ,
                compra IN (televisor, laptop, impresora)
ASSOCIATOR RANGE 2 UNTIL 3
GROUP BY edad, ocupación,compra HAVING count(*)>=2
```

De igual manera se procede con sentencias SQL independientes:

```
SELECT edad, ocupación,compra, count(*) AS soporte INTO ItemsFrecuentes
FROM clientes
EQUIKEEP ON edad IN (20...29,30...40) ,
                ocupación IN (estudiante, ingeniero, medico) ,
                compra IN (televisor, laptop, impresora)
ASSOCIATOR RANGE 2 UNTIL 3
GROUP BY edad, ocupación,compra HAVING count(*)>=2

DESCRIBE MULTIDIMENSIONAL ASSOCIATION RULES
INTO ReglasAsociacion
FROM ItemsFrecuentes
WITH CONFIDENCE 30
LENGTH 3
```

4.2 Primitivas SQL para la tarea de Clasificación.

El operador algebraico *Mate*, conjuntamente con los operadores agregados *Entro* y *Gain*, extienden el álgebra relacional para soportar la tarea de Clasificación. Con el fin de que estos operadores se puedan expresar en el lenguaje SQL, se implementan como primitivas SQL dentro de la cláusula SELECT. El operador algebraico *Describe Classifier* se implementa en el lenguaje SQL como un operador unificado en una cláusula independiente. De esta forma, el lenguaje SQL es capaz de soportar el descubrimiento de reglas de Clasificación (Timarán y Millán, 2006) (Timarán, 2007).

4.2.1 Primitiva *Mate By*

Esta primitiva implementa el operador algebraico *Mate* en la cláusula SQL SELECT. *Mate By* toma los valores de los atributos de una tabla denominados *atributos condición* y por cada registro forma todas las posibles combinaciones de estos atributos con otro atributo de la misma tabla denominado *atributo clase*. Este proceso lo realiza en una sola pasada sobre la tabla.

Dentro de la cláusula SELECT, *Mate By* tiene la siguiente síntaxis:

```
SELECT <ListaAtributosTablaDatos> [INTO <NombreTablaMate>]
FROM <NombreTablaDatos>
WHERE <CláusulaWhere>
MATE BY<ListaAtributosCondicion> WITH <AtributoClase>
GROUP BY <ListaAtributosTablaDatos>

<ListaAtributosTablaDatos> : := <Atributo>,<ListaAtributos>
```

<*ListaAtributos*>: :=<*Atributo*>, <*ListaAtributos*>
<*ListaAtributos*>: :=<*Atributo*>
<*ListaAtributosCondicion*>: :=<*Atributo*>,<*ListaAtributos*>
<*AtributoClase*>::=<*Atributo*>

La primitiva MATE BY facilita la tarea de clasificación y la construcción de un árbol de decisión, al calcular conjuntamente con las funciones agregadas *Gain()* y *Entro()*, en cada partición y para cada atributo, la ganancia de información y la entropía, respectivamente.

Ejemplo 9. Sea la tabla SINTOMAS (SID,D_MUSCULAR,TEMPERATURA, GRIPA) Obtener las ocurrencias de las diferentes combinaciones de los *atributos d_muscular, temperatura* con el atributo *gripa* y almacenar el resultado en la tabla *clasesintomas*. La orden SQL que permite obtener esta consulta es la siguiente:

SELECT *d_muscular,tempeartura,gripa, count(*)* INTO *clasesintomas*
FROM *sintomas*
MATE BY *d_muscular,tempertura* WITH *gripa*
GROUP BY *d_muscular,tempertura,gripa*

4.2.2 Función agregada SQL *Entro ()*

Esta función agregada SQL implementa el operador algebraico agregado *Entro* en la cláusula SQL SELECT. SQL *Entro()* permite calcular, conjuntamente con la primitiva *Mate By*, la entropía de una tabla con respecto a cada una de las combinaciones de los *atributos condición* con el *atributo clase*. SQL *Entro()* se debe ejecutar conjuntamente con la función agregada *Count()*.

Dentro de la cláusula SELECT, SQL *Entro()* tiene la siguiente síntaxis:

SELECT <*ListaAtributosTablaDatos*>, *Count(*)*, *Entro(*)* [INTO <*NombreTablaMate*>]
FROM <*NombreTablaDatos*>
WHERE <*CláusulaWhere*>
MATE BY<*ListaAtributosCondicion*> WITH <*AtributoClase*>
GROUP BY <*ListaAtributosTablaDatos*>

4.2.3 Función agregada SQL *Gain ()*

Esta función agregada SQL, implementa el operador algebraico agregado *Gain* en la cláusula SQL SELECT. SQL *Gain()* permite calcular, conjuntamente con la primitiva *Mate by*, la ganancia de información de una tabla con respecto a cada una de las combinaciones de los *atributos condición* con el *atributo clase*. Internamente SQL *Gain()* calcula la entropía del *atributo clase*, por ello se debe ejecutar conjuntamente con las funciones agregadas *Count()* y *Entro()*.

Dentro de la cláusula SELECT, SQL *Gain ()* tiene la siguiente síntaxis:

SELECT <*ListaAtributosTablaDatos*>, *Count (*)*, *Entro (*), Gain (*)*
[INTO <*NombreTablaMate*>]
FROM <*NombreTablaDatos*>

WHERE <*CláusulaWhere*>
MATE BY<*ListaAtributosCondicion*> WITH <*AtributoClase*>
GROUP BY <*ListaAtributosTablaDatos*>

4.2.4 Operador Unificado SQL *Describe Classification Rules*

El operador SQL *Describe Classification Rules* implementa el operador algebraico *Describe Classifier* en una nueva cláusula SQL. Este operador construye el árbol de decisión y genera las reglas de clasificación. El operador *Describe Classification Rules* permite la construcción del árbol de decisión de manera unificada con el cálculo de la métrica de particionamiento con la primitiva *Mate By* y las funciones agregadas *Entro()* y *Gain()* en una sola instrucción SQL o en forma separada con sentencias SQL independientes, para luego generar las reglas.

Describe Classification Rules tiene la siguiente sintaxis:

DESCRIBE CLASSIFICATION RULES
[INTO <*TablaReglasClasificación*>]
FROM <*NombreTablaArbol*>
USING <*NombreTablaMétrica*>
[DO <*SubconsultaCálculoMétrica*>]

<*SubconsultaCálculoMétrica*>::=<SFWMG>
<SFWMG> ::= <SELECT FROM WHERE MATE BY GROUP BY>

Ejemplo 10. Sea la tabla CLIENTES (EDAD, INGRESOS, ES_ESTUDIANTE, MANEJOCREDITO,COMPRAEQUIPO) cuyos valores han sido discretizados. Encontrar las reglas de clasificación y almacenarlas en la tabla ReglasClasificación.

La sentencia SQL unificada que genera las reglas de clasificación es:

DESCRIBE CLASSIFICATION RULES
INTO *reglasclasificación*
FROM *nodosarbol*
USING *métricaspartición*
DO SELECT *edad, ingresos, es_estudiante, manejocredito, compraequipo,*
count(*), Entro() AS Entropia, Gain() AS Ganancia
INTO *métricaspartición*
FROM *clientes*
MATE BY *edad, ingresos, as_estudiante, manejocredito* WITH *compraequipo*
GROUP BY *edad, ingresos, es_estudiante, manejocredito, compraequipo*

De igual manera se procede con sentencias SQL independientes:

SELECT *edad, ingresos, es_estudiante, manejocredito, compraequipo,*
count(*), Entro() AS Entropia, Gain() AS Ganancia
INTO *métricaspartición*
FROM *clientes*
MATE BY *edad, ingresos, as_estudiante, manejocredito* WITH *compraequipo*
GROUP BY *edad, ingresos, es_estudiante, manejocredito, compraequipo*

DESCRIBE CLASSIFICATION RULES
INTO *reglasclasificación*
FROM *nodosarbol*
USING *métricaspartición*

En este ejemplo a partir de la tabla *métricaspartición*, el operador *Describe Classification Rules* construye la tabla *nodosarbol* y con ella genera las reglas, almacenándolas en la tabla *reglasclasificación.*

5 Aspectos de implementación de PostgresKDD en un acoplamiento fuerte con el SGBD PostgreSQL

Para el desarrollo de PostgresKDD se utilizó un equipo Intel Pentium IV de 64 bits a 3,0 Ghz, memoria RAM de 2Gb, disco duro SERIAL ATA DE 160 Gb, Sistema Operativo FEDORA CORE 5. La versión de PostgreSQL modificada fue la 7.3.4.

5.1 Operadores en PostgreSQL.

Cuando una consulta se ejecuta en Postgres (Stonebraker y Rowe, 1986), implícitamente se hace referencia a Operadores como seleccionar, ordenar y agrupar, entre otros. El *Planner/Optimizer* recibe la consulta y la transforma en un plan de ejecución, que se envia al *Executor* para su procesamiento.

Si el plan entregado por el *Planner/Optimizer* al *Executor* contiene un solo nodo, lo que significa que este nodo es equivalente al Operador, en este caso, se llama "Operador Sencillo", ejemplo: Order by, Result, Append, SeqScan, IndexScan.

En el caso que el plan asociado a este operador esté compuesto por varios nodos, se denomina "Operador Complejo”**,** ejemplo: Group by, MergeJoin, HashJoin, Agg, NestLoop, Hash.

5.1.1 Un operador sencillo.

El Operador *SeqScan*, es un operador sencillo que recorre secuencialmente una tabla en una dirección determinada. Este es un Operador Sencillo debido a que Postgres lo representa con un solo nodo en el plan de ejecución. Este operador al igual que el resto de los Operadores de Postgres, tiene tres rutinas de interfaz asociadas a cada un de las fases del Executor: ExecInitScan, ExecSeqScan y ExecEndScan.

Dentro de la Rutina ExecSeqScan, donde se realiza la ejecución del Operador, se encuentra la función SeqNext, la cual retorna la tupla, con el apoyo de información como la dirección de la búsqueda, el estado común de búsqueda y el estado de la consulta. Esta a su vez captura la tupla a retornar por medio de la función Heap_GetNext, la cual toma como parámetro la información actual de la búsqueda, la dirección y el buffer donde será

almacenada la tupla. La tupla capturada se guarda haciendo uso de la función ExecStoreTuple.

Ejemplo 11. Sea la Tabla VENDEDORES (ID,NOMBRE,CIUDAD). Realizar una consulta y observar el nodo formado por el Operador *SeqScan* (Nodo SeqScan) en la figura 16.

SELECT *nombre, ciudad*
FROM *vendedores*
WHERE *id >2*

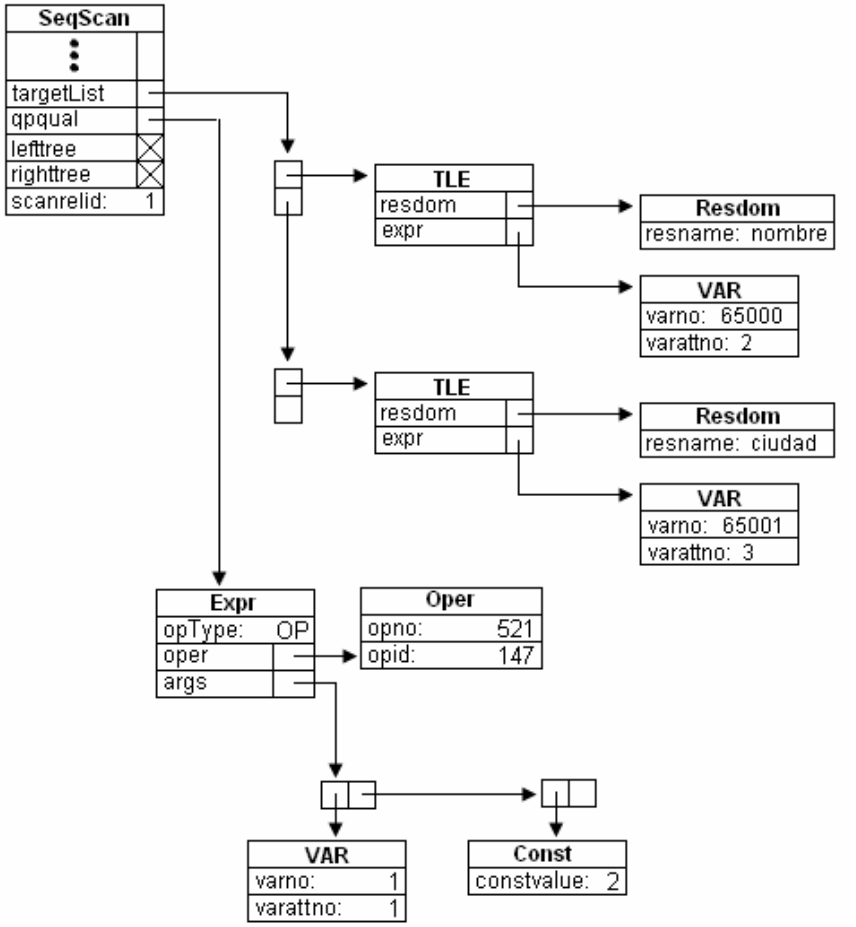

Figura 16. Plan de ejecución de un Operador Sencillo (SeqScan)

En la Figura 16 se observa el plan de ejecución de un Operador Sencillo. El *Executor* llama recursivamente al plan presente en la cima del nodo *SeqScan*. Los campos *lefttree* y *righttree* presente en el nodo *SeqScan* no contienen apuntadores a subplanes, ellos están en NULL. El campo *targetList* contiene un lista de apuntadores a nodos TLE, en el cual se

encuentran presentes dos campos, el primero *resdom*, es un apuntador a un nodo de tipo *Resdom* el cual contiene el nombre del atributo (*resname*), el segundo campo es un apuntador a un nodo de tipo *Var* el cual contiene la posición del atributo en la Tabla *Vendedores* (2 y 3), así como el identificador dado a cada atributo internamente por Postgres. El campo *qpqual* contiene un apuntador a un nodo *Expr*, en el cual se encuentran presente tres campos, el primero *opType*, contiene el operador relacional presente en la consulta (>,<,=,…), el segundo es un apuntador a un nodo de tipo *Oper*, el cual contiene el identificador asignado al operador en Postgres, el tercer campo es un apuntador a dos nodos, uno de tipo *Var* el cual contiene la posición del atributo en la tabla, y el nodo *Const* contiene el valor constante presente en la condición (*where id>2*). El campo *scanrelid* contiene un valor que identifica la existencia de índices en la tabla presente en la consulta.

5.1.2 Un operador Complejo.

El Operador de Agregado *(Agg)* está diseñado para manejar consultas que contenga subconsultas con el Operador *Group by* (GRP) y a su vez el Operador Group by es diseñado para manejar consultas que contenga el *Operador Sort* y el *Operador SeqScan*. Se asume que las tuplas resultantes del plan de salida están ordenadas según las especificaciones de las columnas del Group by, es decir, las tuplas de un mismo grupo están consecutivas. De esta manera, se muestra que el Operador *Agg* se representa por tres Operadores en el plan de Ejecución: *Group by*, *Sort*, *SeqScan*.

Ejemplo 12. Sean la tabla VENTAS (ID, CANTIDAD, VALOR). Realizar una consulta y observar el plan de ejecución formado por el Operador Complejo (*Nodo Agg*) en la figura 17.

```
SELECT      id, count(cantidad)
FROM        ventas
GROUP BY    id
```

En la Figura 17 se observa el plan de ejecución de un Operador Complejo. El *Executor* llama recursivamente al plan presente en la cima del nodo Agregado *(Agg).* El campo *lefttree* presente en el nodo *Agg* contiene un apuntador a un subplan, el primer nodo del subplan es un *Group by* que a su vez contiene un apuntador a un *nodo Sort*. El nodo *Sort* contiene como subplan un nodo *SeqScan.*

Todos estos subplanes son llamados recursivamente empezando con el nodo *Agg* y terminando con el nodo *SeqScan*. El campo *qptargetlist* contiene un lista de apuntadores a nodos TLE, en el cual se encuentran presentes dos campos, el primero *resdom*, es un apuntador a un nodo de tipo *Resdom* el cual contiene el nombre del atributo *(resname*), el segundo campo es un apuntador a un nodo de tipo *Var* el cual contiene la posición del atributo en la Tabla *Ventas* (1 y 2), así como el identificador dado a cada atributo internamente por Postgres. El campo *aggs* contiene un apuntador a un nodo *Aggreg*, en el cual se encuentran presente dos campos, el primero *aggname* que contiene el nombre de la función agregada, utilizada en la consulta (*count*) y el segundo es un apuntador a un nodo de tipo *target.*

5.1.3 Estructura general de un operador de PostgreSQL.

Postgres clasifica los Operadores según su tarea (agrupar, ordenar, contar, entre otros), estos se encuentran el archivo *.../include/nodes/plannodes.h.* En el nivel superior del plan de ejecución se encuentran nodos como, *Append* utilizado en consultas de actualización, *Result* para consultas de selección, *SeqScan* para extraer tuplas, *IndexScan* para tuplas en tablas referenciadas con Índices, entre otros.

En Postgres un nodo (Operador) describe una estructura con parámetros generales que todos obtienen y parámetros particulares de acuerdo al tipo de nodo. En la Figura 18 se muestra el Tipo Abstracto de Datos T.D.A. de un nodo *SeqScan*.

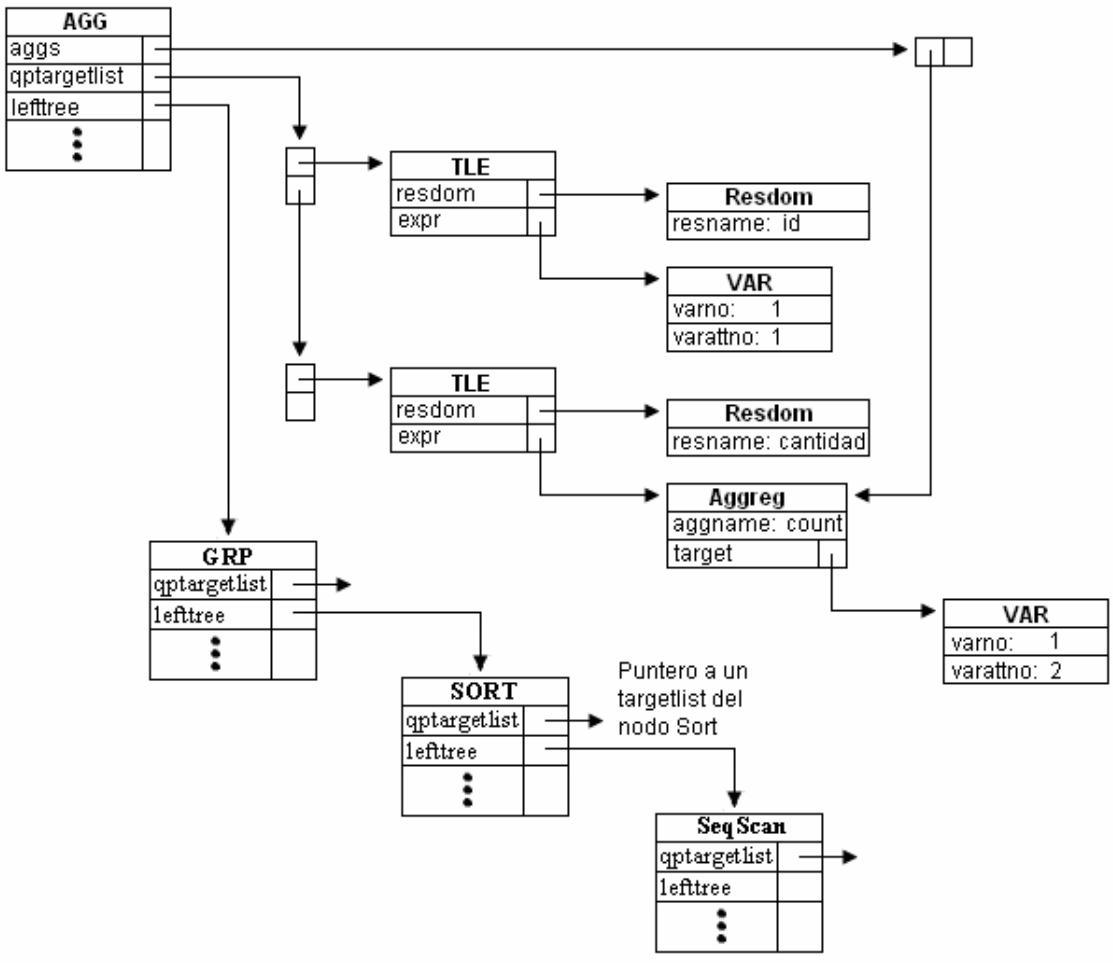

Figura 17. Plan de ejecución de un Operador Complejo (Agg)

```
typedef struct Scan
{
        Plan                    plan;
        índex                   scanrelid;
        CommonScanState         scanstate;
} Scan;
```

Figura 18. T.D.A de un nodo SeqScan

En el T.D.A de la Figura 18 se puede identificar tres parámetros principales:

- **El Plan**

Este parámetro es un campo de tipo *Plan*. El tipo *Plan* está definido en el archivo *../src/include/nodes/plannodes.h*. El *plan* es el parámetro más importante de todos, debido a que contiene la información básica de los nodos. Es utilizado por el *Executor* en todas las fases. En la figura 19 se muestra la definición de esta estructura.

```
typedef struct Plan
{
        Node Tag                type;
        Cost                    startup_ cost;
        Cost                    total cost;
        double                  plan_ rows;
        int                     plan_with;
        Estate                  *state;
        List                    *targetlist;
        List                    *qual;
        Struct Plan             *lefttree;
        Struct Plan             *righttree;
        List                    *extParam;
        List                    *locParam;
        List                    *chgParam;
        List                    *initPlan;
        List                    *subPlan;
        Int                     *ParamExec;
}Plan;
```

Figura 19. T.D.A de un Plan de Ejecución

Los elementos de esta estructura son:

• type (NodeTag): Representa el tipo del Nodo. Cada nodo tiene un tipo o identificador, que le permite al *Executor* identificarlo de los demás, y así llamar a su correspondiente rutina de

manipulación. El tipo de nodo está directamente asociado a un tipo *enum*, llamado *NodeTag*, definido en el archivo *.../src/include/nodes/node.h.* El tipo *NodeTag* contiene el conjunto de todos los posibles nodos que manipula el sistema.

• startup_cost (Cost): representa el costo estimado del tiempo de ejecución de las operaciones a realizar por el nodo. Este dato es utilizado por el optimizador para escoger la mejor forma de satisfacer una consulta determinada.

• total_cost (cost): representa el costo total del tiempo de ejecución del plan de ejecución escogido.

• state (Estate): describe la información general del estado actual del plan de ejecución o consulta al cual pertenece el nodo. El estado de la consulta guarda información relacionada con: el conjunto de relaciones que intervienen en la consulta, los bloques de memoria disponibles, y la dirección de la búsqueda, entre otros. Este campo es igual para todos los nodos de un mismo plan de ejecución.

• ríghttree, lefttree (Struct Plan): representa los planes de entrada y salida del nodo. El plan de entrada representa el origen de las tuplas a procesar por dicho nodo y el plan de salida representa el destino de las tuplas generadas por este.

Existen otros campos no menos importante como: el tamaño y el ancho estimado de la relación, el número máximo de tuplas por página y la lista de subplanes que representan subconsultas, entre otros.

- **Estado de un Nodo**

El estado contiene la información necesaria para retornar la siguiente tupla en el momento que el *Executor* lo requiera. Este es utilizado por las funciones de interfaz del nodo para llevar a cabo esta labor. El estado del nodo es una estructura de tipo *CommonEstate* incluida en el archivo *.../src/include/nodes/execnodes.h*. La figura 20 presenta la definición de la estructura *CommonEstate.*

```
typedef struct CommonState
{
        NodeTag             type;
        Int                 cs_base_id;
        TupleTableSlot      *cs_OuterTupleSlot;
        TupleTableSlot      *cs_ResultTupleSlot;
        ExprContext         *cs_ExprContext;
        Projectioninfo      *cs_ProjInfo;
        bool                cs_TupFromTlist;
} CommonState;
```

Figura 20. T.D.A del estado general de un nodo

Sus principales campos son:

• type (NodeTag): El tipo *NodeTag* contiene el conjunto de todos los posibles nodos que manipula el sistema, como por ejemplo, *CommonState*.

• cs_base_id (int) (Identificador de nodos): Es un identificador único asignado por el planeador en la fase de inicialización, y que lo diferencia de los demás estados de nodos. Es como un número de proceso asignado a una tarea en los sistemas operativos.

• cs_OuterTupleSlot (Tuple TableSlot) (Tupla resultante, sin proyectar): Apuntador a la tupla de resultante sin realizar la proyección correspondiente al nodo. La rutina de ejecución del nodo calcula esta tupla.

• cs_ResultTupleSlot (TupleTableSlot) (Tupla resultante proyectada): Apuntador a la tupla resultante con la proyección correspondiente al nodo. Resulta de proyectar cs_Outer_TuipleSlot. Esta tupla se retorna al nodo padre o plan de salida.

• cs_ExprContext (ExprContext) (Expresión actual del contexto): Contiene la información del contexto (Bloque de memorias) necesaria para clasificar y proyectar las tuplas resultantes.

• cs_ProjInfo (proyectioInfo) (Proyección): Indica que proyección se debe realizar a la tupla de salida, para obtener la tupla resultante.

Algunos nodos necesitan información adicional, como en el caso del nodo *SeqScan*, que se utiliza cuando se hacen consultas de selección. Para estos se crean estructuras especializadas, que contienen la información adicional requerida. En el estado del nodo *SeqScan* (figura 21), se encuentra la información básica del estado (campo cstate de tipo CommonState), y una información adicional (campos css_currentRelation, csscurrent y css_ScanTupleSlot) para indicar que la tarea del nodo ya se terminó.

```
typedef struct CommonScanState
{
        CommonState     estáte;
        Relation        css_currentRelation;
        HeapScanDesc    css_currentScanDesc;
        TupleTableSlot  *css_ScanTupleSlot;
} CommonScanState;
```

Figura 21. T.D.A del estado del nodo SeqScan

Al crear un nuevo nodo, se debe establecer que información de estado se requiere. En el caso que la estructura *CommonEstate* sea suficiente para almacenar la información puede no será necesario crear una nueva estructura derivada.

- **Interfaz de un Nodo**

La interfaz de un nodo hace referencia a las rutinas de manipulación del mismo. Estas realizan las tareas específicas que se asocian a cada nodo. Contienen tres funciones que corresponden a las fases del *Executor*: inicializar (*Execinitnodo*), ejecutar (*Execnodo*), y finalizar (*ExecEndnodo*). Además existe una rutina *ExecCountSlotsnodo* la cual calcula el número de Slots (tuplas) que utiliza el nodo. Estas rutinas se definen en el archivo *.../src/include/executor/.*

5.2 Implementación de la primitiva *Associator Range* al interior del motor de PostgreSQL

El proceso de implementación de la primitiva *Associator Range* contempla la modificación de las estructuras, funciones y la creación de nuevos nodos en las etapas que componen la capa intermedia de la arquitectura de Postgres:

- El Parser ha sido modificado para que construya, transforme y adjunte a las estructuras del compilador una lista de números enteros para *Associator Range*.

- El Planner/Optimizer recibe el *parser tree*, verifica si lista de *Associator Range* contiene valores, en cuyo caso agrega un nodo *Associator* al *Queryplan.*

- El Executor ha sido modificado para evaluar el nuevo nodo y entregar un conjunto de tuplas, donde la cantidad de registros generados depende de los valores asignados a la lista *Associator Range*.

En la modificación de programas se debe tener en cuenta que solamente se presentan las partes relevantes de código afectado. Cada línea de código agregada será marcada por un '+' al principio de la línea y cada línea de código modificada será marcada con un '!' a través de los siguientes listados de código.

5.2.1 Parser.

El análisis léxico no se ha modificado pues no se han introducido nuevos símbolos.

Para el análisis sintáctico fue necesario introducir cambios y nuevo código en las funciones o estructuras de los siguientes archivos:

- Con el fin de introducir las nuevas palabras reservadas *Associator y Range* se modifica el archivo *…/src/backend/parser/keywords.c* donde las palabras reservadas se listan en la estructura *ScanKeywords* , manteniendo un estricto orden alfabético.

- En el archivo *…/src/backend/parser/gram.y* se adiciona las nuevas reglas de producción siguiendo la estructura lógica de este archivo especial de gramática para hacer funcional las nuevas producciones sin alterar las existentes, tal como muestra la figura 22.

```
+       associator_clause:
+               ASSOCIATOR_P RANGE associator_list      {$$ = $3;}
+               | /*EMPTY*/                             {$$ = NIL;}
+               ;

+       associator_list:
+               Iconst                                  { $$ = makeListi1($1); }
+               | associator_list UNTIL Iconst                  { $$ = lappendi($1, $3); }
+               ;
```

Figura 22. Nuevas reglas de Producción

Ahora, en la regla *simple_select* se establece el punto para activar las nuevas reglas de producción (figura 23).

```
        simple_select:
                SELECT opt_distinct target_list
                into_clause from_clause where_clause
!               associator_clause group_clause having_clause
                        {
                                SelectStmt *n = makeNode(SelectStmt);
                                n->distinctClause = $2;
                                n->targetList = $3;
                                n->into = $4;
                                n->intoColNames = NIL;
                                n->fromClause = $5;
                                n->whereClause = $6;
+                               n->associatorClause = $7;
                                n->groupClause = $8;
                                n->havingClause = $9;
                                $$ = (Node *)n;
                        }
```

Figura 23. Regla simple_select modificada.

- Para recibir y almacenar los valores de rango inicial y final del operador *Associator*, fué necesario agregar una lista a la estructura *SelectStmnt (…/src/include/nodes/parsenodes.h*). La figura 24 presenta la estructura *SelectStmnt* modificada.

```
        typedef struct SelectStmt
        {       NodeTag             type;

                List        *distinctClause;
                RangeVar    *into;              /* target table (for select into table) */
```

```
                List    *intoColNames;    /* column names for into table */
                List    *targetList;      /* the target list (of ResTarget) */
                List    *fromClause;              /* the FROM clause */
                Node    *whereClause;             /* WHERE qualification */
+               List    *associatorClause;        /* lista de ASSOCIATOR RANGE */
                List    *groupClause;             /* GROUP BY clauses */
                Node    *havingClause;    /* HAVING conditional-expression */
                ...
        } SelectStmt;
```

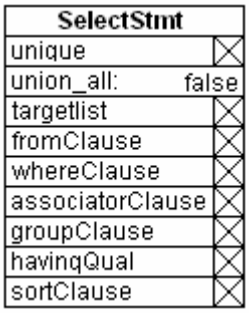

Figura 24. SelectStmnt modificada

Para continuar normalmente la etapa de transformación (transformar el *parser tree* a un *nodo Query*) fue necesario realizar las siguientes modificaciones:

- Para llevar los datos almacenados de la estructura *SelectStmnt* a un nodo *Query*, se creo la función *transformAssociatorClause*, cuya definición esta en el archivo *…/src/include/parser/parser_clause.h* y se implementó físicamente en *…/src/backend/parser/parser_clause.c* (figura 25).

```
+       List *
+       transformAssociatorClause(ParseState *pstate,
+                                 List *associatorClause,
+                                 List *targetlist)
+       {
+               List       *alist = NIL;

+               if (length(associatorClause)==0 || length(associatorClause)!=2)
+                       elog(ERROR, "Numero de parámetro en AC incorrectos");
+               else
+               {
+                       if(nthi(0,associatorClause) > nthi(1,associatorClause))
+                               elog(ERROR, "Primer parámetro de AC debe ser el
                                        menor");
+                       else if (nthi(1,associatorClause) > length(targetlist))
                                elog(ERROR,
+                               "Segundo parámetro de AC es mayor que len TL");
```

```
+                       else
+                               alist = lappendi(alist,nthi(0,associatorClause));
+                               alist = lappendi(alist,nthi(1,associatorClause));
+               }

+               return alist;
+       }
```

Figura 25. Función transformAssociatorClause

- Se modificó el nodo Query *(…/src/include/nodes/parsenodes.h*) con el fin de que pueda recibir y almacenar los datos de la estructura *SelectStmnt* que pertenecen al nuevo operador. La figura 26 presenta el nodo *Query* modificado.

- La función *transformSelectStmt* del archivo *…/src/backend/parser/analize.c* es la encargada de iniciar la transformación de las estructuras, por tal razón fue necesario incluir en este punto, una referencia a la función *transformAssociatorClause*.

```
typedef struct Query
{
        NodeTag     type;
        CmdType     commandType;
        ...
        List        *rtable;        /* list of range table entries */
        FromExpr    *jointree;      /* table join tree (FROM and WHERE *
                                       clauses) */
        List        *rowMarks;      /* integer list of RT indexes of relations
                                       that are selected FOR UPDATE */
        List    *targetList;  /* target list (of TargetEntry) */
+       List    *associatorClause;      /* una lista para
                                           AssociatorClause */
        List    *groupClause;           /* a list of GroupClause's */
        Node    *havingQual     /* qualifications applied to groups */
        List    *distinctClause;        /* a list of SortClause's */
        List    *sortClause;            /* a list of SortClause's */
        ...
} Query;
```

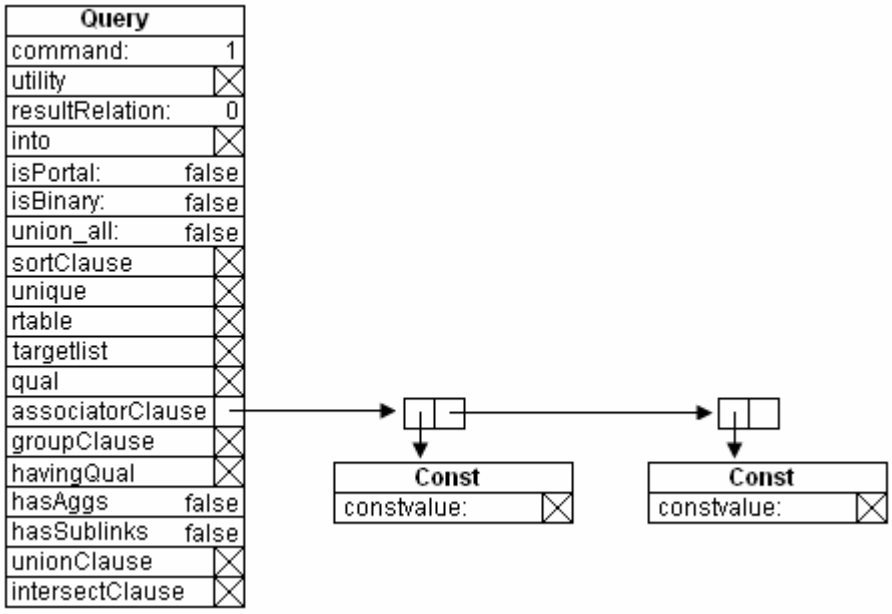

Figura 26. Query modificado

5.2.2 Rewriter.

Puesto que no se contempla la implementación de las primitivas de Asociación y Clasificación como parte de la definición de vistas, ni reglas de transformación a partir de estas, no se realizó ninguna modificación a los módulos que hacen esta tarea.

5.2.3 Planner/Optimizer.

Esta etapa es la encargada de tomar la información del nodo *Query* para crear un plan de ejecución. Para adicionar el nuevo operador al plan de ejecución fue necesario realizar las siguientes modificaciones:

- Con el fin de reconocer el nuevo operador en la etapa de optimización, primero se asigna una etiqueta (*T_Associator*) en la sección PLAN NODES del archivo *…/src/include/nodes/nodes.h* y segundo, se define la estructura de datos para este operador en el archivo *…/src/include/nodes/plannodes.h*. La figura 27 presenta el nuevo nodo para *Associator*.

```
+       typedef struct Associator
+       {
+               Plan            plan;
+               int             inicio;
+               int             fin;
+               AssociatorState *astate;
+       } Associator;
```

Figura 27. Nodo Associator

- Para identificar la nueva estructura *AssociatorState* primero se adiciona la etiqueta *T_AssociatorState* en el archivo *…/src/include/nodes/nodes.h* y después se define su estructura en el archivo *src/include/nodes/execnodes.h* como lo muestra la figura 28. *AssociatorState* es la encargada de llevar la información necesaria para retornar o almacenar las tuplas a través de las funciones de interfaz de nodo cuando el *Executor* lo requiera

```
+       typedef struct AssociatorState
+       {
+               CommonScanState     csstate;
+               bool                pedir_tupla;
+               bool                retornar_tupla;
+               int                 num_tuplas;
+               int                 out_tuplas;
+               int                 rginicio;
+               int                 rgfin;
+               TupleTableSlot      *tupla;
+               void                *tuplestorestate;
+       } AssociatorState;
```

Figura 28. AssociatorState

Los campos presentes en la estructura AssociatorState son:

- csstate (CommonScanState): Representa la información básica del estado en que se encuentra el nodo.
- pedir_tupla (bool): Utilizado para que el nodo SeqScan retorne o no una tupla.
- retornar_tupla (bool): Utilizado para que el nodo superior reciba o no las tuplas generadas.
- num_tuplas (int): Cuenta las tuplas generadas por el algoritmo.
- out_tuplas (int): Cuenta las tuplas generadas por el algoritmo que han sido entregadas al nodo superior.
- rginicio, rgfin (int): Son los parámetros dados al operador Associator range. Controlan el rango de inicio y de fin de la asociación.
- *tupla (TupleTableSlot): Almacena la tupla pasada por el nodo SeqScan.
- *tuplestorestate (tuplestorestate): Almacena las tuplas generadas por el algoritmo para después pasarlas al nodo superior.

- Para crear el nuevo nodo *Associator* con los datos almacenados en el *Query* y los valores actuales del costo de ejecución, se define la función *make_Associator* en el archivo *…/src/include/optimizer/planmain.h* y se implementa físicamente en el archivo *…/src/backend/optimizer/plan/createplan.c* (figura 29).

```
+       Associator *
+       make_associator(List *qptlist,List *alist, Plan *lefttree)
+       {
+               Associator    *node = makeNode(Associator);
+               Plan      *plan = &node->plan;

+               copy_plan_costsize(plan, lefttree);

+               plan->state = (EState *) NULL;
+               plan->targetlist = qptlist;
+               plan->qual = NIL;
+               plan->lefttree = lefttree;
+               plan->righttree = NULL;

+               node->inicio = nthi(0,alist) ;
+               node->fin= nthi(1,alist);
+               node->astate = (AssociatorState *) NULL;

+               return node;
+       }
```

Figura 29. Función make_associator

- La función *grouping_planner* del archivo …*/src/backend/optimizar/plan/planner.c* es la encargada de asignar el orden de los nodos en el plan de ejecución, por tal razón es necesario incluir en este punto, una referencia a la función *make_associator* como lo muestra la figura 30.

```
        static Plan *
        grouping_planner(Query *parse, double tuple_fraction)
        {
        …
+       if (parse->associatorClause)
+       {
+               List        *alist;
+
+               if (parse->hasAggs)
+                       alist = new_unsorted_tlist(result_plan->targetlist);
+               else
+                       alist = tlist;
+               result_plan = (Plan *) make_associator(alist,
+                                                      parse->associatorClause,
+                                                      result_plan);
+       }
        …
        }
```

Figura 30. Función grouping_planner modificado

- Por último en la función *set_plan_references* del archivo *…/src/backend/optimizer/plan/setrefs.c* se incluye la etiqueta *T_Associator* para el proceso final de la etapa *planner/optimizer*, se complete el plan de ejecución y se realicen los ajustes necesarios para que el *Executor* pueda trabajar adecuadamente. La figura 31 presenta el plan de ejecución de la primitiva *Associator Range* en un operador complejo.

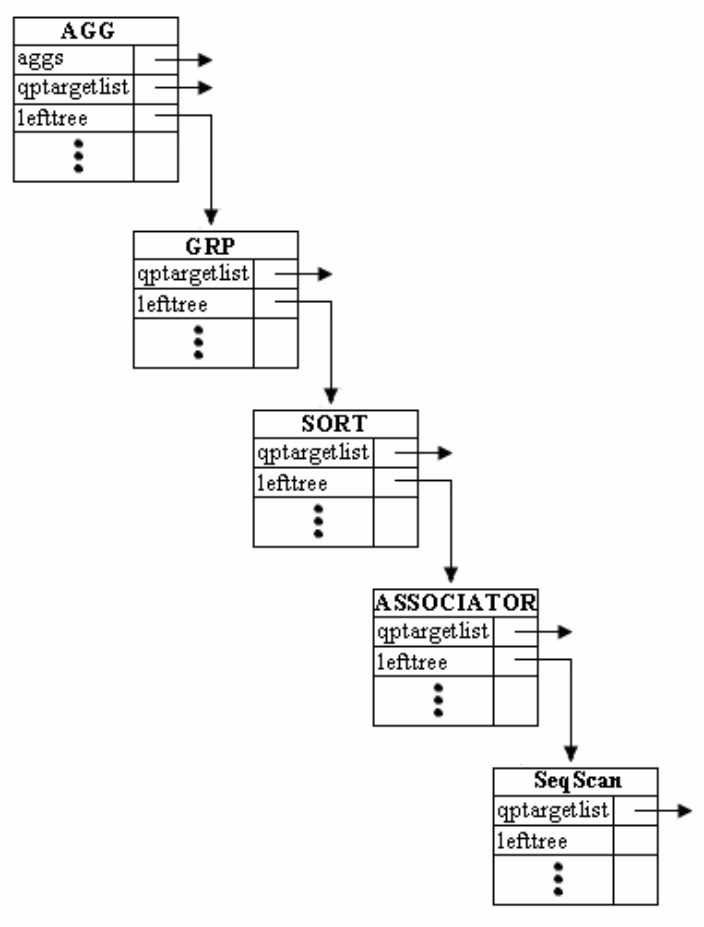

Figura 31. Plan de Ejecución de Associator Range

5.2.4 Executor.

La interfaz de un nodo hace referencia a las rutinas de manipulación del mismo. Estas rutinas realizan las tareas específicas de cada nodo. Contiene tres funciones que corresponden a las fases del Ejecutor: inicializar (*ExecInitAssociator*), ejecutar *(ExecAssociator)* y finalizar (*ExecEndAssociator*). En esta etapa se realiza las siguientes modificaciones:

- Para definir las funciones de manipulación del nodo *Associator* se crean los archivos *…/src/include/executor/nodeAssociator.h* y *…/src/backend/executor/nodeAssociator.c*

para la implementación de las mismas. La figura 32 presenta las funciones de manipulación del nodo Associator.

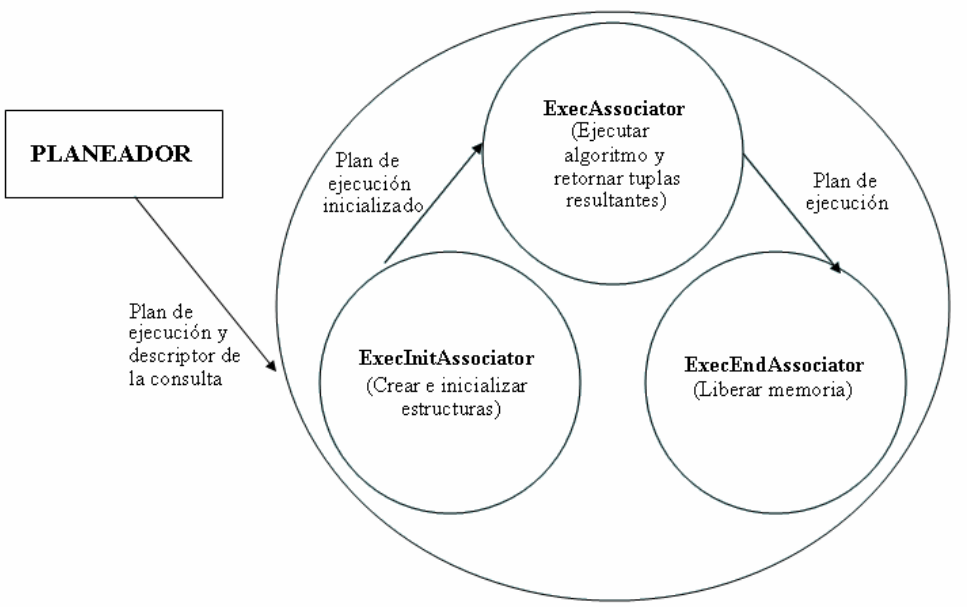

Figura 32. Funciones de manipulación del nodo Associator

- La función encargada de realizar la fase de inicialización del nodo *Associator* es *ExecutorStart*, declarado en el archivo *.../src/backend/executor/execMain.c*. *ExecutorStart* llama a la función *InitPIan*, declarada en el mismo archivo. Esta función inicializa el plan de consulta y llama a la función *ExecInitNode*, declarada en el archivo *.../src/backend/executor/execProcNode.c*, en el cual se adicionó *ExecInitAssociator* (figura 33).

```
        bool
        ExecInitNode(Plan *node, EState *estate, Plan *parent)
        {
                ...
                switch (nodeTag(node))
                {
                 ...
+                       case T_Associator:
+                               result = ExecInitAssociator((Associator *)
                                        node, estate, parent);
+                       break;
                ...
                }
        }
```

Figura 33. Función ExecInitNode modificado

- La función *ExecutorRun* declarada en el archivo *.../src/backend/executor/execMain.c*, es la encargada de realizar la fase de ejecución del plan generado en el *Planner/Optimizer* y llama a la función *ExecutePlan*, declarada en el mismo archivo. Esta función procesa el plan de ejecución y retorna el número total de tuplas como las tuplas mismas e invoca a *ExecProcNode* declarada en el archivo *.../src/backend/executor/execProcNode.c*, donde se adicionó *ExecAssociator* (Figura 34).

```
TupleTableSlot *
ExecProcNode(Plan *node, Plan *parent)
{
        …
        switch (nodeTag(node))
        {
         …
+               case T_Associator:
+                       result = ExecAssociator((Associator *) node);
+                       break;
        …
        }
}
```

Figura 34. Función ExecProcNode modificado

- Al final de la ejecución del plan, la función *ExecutorEnd* declarada en el archivo *.../src/backend/executor/execMain.c*, se encarga de liberar todos los recursos que se reservaron en su ejecución. La función *ExecutorEnd* invoca la función *EndPlan*, del mismo archivo. Esta función finaliza el plan de ejecución (Cerrar archivos y liberar memoria de estructuras) e invoca a *ExecEndNode*, declarada en el archivo *.../src/backend/executor/execProcNode.c.* donde se adicionó *ExecEndAssociator* (figura 35).

```
void
ExecEndNode(Plan *node, Plan *parent)
{
        …
        switch (nodeTag(node))
        {
                …
+               case T_Associator:
+                       ExecEndAssociator((Associator *) node);
+                       break;
                …
        }
}
```

Figura 35. Función ExecEndNode modificado

5.3 Implementación de la primitiva *EquiKeep On* al interior del motor de PostgreSQL

Para la implementación del operador *EquiKeep* y la primitiva *Equikeep On*, al igual que con Associator, se deben modificar las estructuras, funciones y crear nuevos nodos en las etapas que componen la capa intermedia de la arquitectura de Postgres:

- El Parser ha sido modificado para que construya, transforme y adjunte a las estructuras del compilador una lista de expresiones lógicas que necesita éste operador.

- El Planner/Optimizer recibe el parser tree, cuantifica y transforma (Forma Normal Conjuntiva CNF, Forma Normal Disyuntiva DNF) el conjunto de expresiones lógicas de éste operador y agrega un nodo *Equikeep* al *Queryplan*.

- El Executor ha sido modificado para restringir los valores de los atributos de cada una de las tuplas a únicamente los valores de los atributos que satisfacen la expresión lógica correspondiente.

5.3.1 Parser.

Al igual que el operador anterior, no modifica el análisis léxico ya que no se agregan nuevos símbolos.

Para el análisis sintáctico se introducen cambios y nuevo código en las funciones o estructuras de los archivos que se mencionan adelante.

- Con el fin de introducir las nuevas palabras reservadas *Equikeep y On* se modifica el archivo …*/src/backend/parser/keywords.c* donde las palabras reservadas se listan en la estructura ScanKeywords [], manteniendo un estricto orden alfabético.

- En el archivo …*/src/backend/parser/gram.y* se adiciona las nuevas reglas de producción siguiendo la estructura lógica de este archivo especial de gramática para hacer funcional las nuevas producciones sin alterar las existentes, tal como muestra la figura 36.

```
+               equikeep_clause:
+                       EQUIKEEP keep_list  { $$ = $2; }
+                       | /*EMPTY*/                 { $$ = NIL; }
+                       ;
+               keep_list:
+                       a_expr                      { $$ = makeList1($1); }
+                       | keep_list ',' a_expr{ $$ = lappend($1, $3); }
+                       ;
```

Figura 36. Nuevas reglas de Producción

Ahora, en la regla *simple_select* se establece el punto para activar las nuevas reglas de producción (figura 37).

```
simple_select:
            SELECT opt_distinct target_list
            into_clause from_clause where_clause
!           equikeep_clause group_clause having_clause
                    {
                        SelectStmt *n = makeNode(SelectStmt);
                        n->distinctClause = $2;
                        n->targetList = $3;
                        n->into = $4;
                        n->intoColNames = NIL;
                        n->fromClause = $5;
                        n->whereClause = $6;
+                       n->equikeepClause = $7;
                        n->groupClause = $8;
                        n->havingClause = $9;
                        $$ = (Node *)n;
                    }
```

Figura 37. Regla simple_select modificada.

- Para recibir y almacenar los valores el conjunto de expresiones lógicas del operador *Equikeep*, se agregó una lista a la estructura *SelectStmnt (…/src/include/nodes/parsenodes.h*). La figura 38 presenta la estructura *SelectStmnt* modificada.

```
typedef struct SelectStmt
{       NodeTag         type;

        List    *distinctClause;
        RangeVar  *into;          /* target table (for select into table) */
        List    *intoColNames;    /* column names for into table */
        List    *targetList;      /* the target list (of ResTarget) */
        List    *fromClause;            /* the FROM clause */
        Node    *whereClause;           /* WHERE qualification */
+       List    *equikeepClause;/* lista de Equikeep */
        List    *groupClause;           /* GROUP BY clauses */
        Node    *havingClause;    /* HAVING conditional-expression */
        ...
} SelectStmt;
```

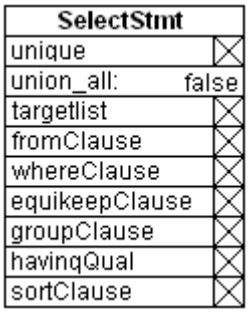

Figura 38. SelectStmnt modificada

Para continuar normalmente la etapa de transformación (transformar el *parser tree* a un nodo *Query*) fue necesario realizar las siguientes modificaciones:

- Para llevar los datos almacenados de la estructura *SelectStmnt* a un nodo *Query*, se creo la función *transformEquikeepClause*, cuya definición esta en el archivo *…/src/include/parser/parser_clause.h* y se implementó físicamente en *…/src/backend/parser/parser_clause.c* (figura 39).

```
+       List *
+       transformEquikeepClause(ParseState *pstate,List *keeplist)
+       {
+               List *lst;
+               List *newkeep;
+
+               newkeep = NIL;
+               foreach(lst, keeplist)
+               {
+                       Node  *transkqual;
+                       Node  *kqual = (Node *) lfirst(lst);
+
+                       transkqual =  transformWhereClause(pstate, kqual);
+
+                       newkeep = lappend(newkeep, transkqual);
+               }
+               return newkeep;
+       }
```

Figura 39. Funcion transformEquikeepClause

- Se modificó el nodo *Query* (*…/src/include/nodes/parsenodes.h*) con el fin de que pueda recibir y almacenar los datos de la estructura *SelectStmnt* que pertenecen al nuevo operador. La figura 40 presenta el nodo *Query* modificado.

- La función transformSelectStmt del archivo *…/src/backend/parser/analize.c* es la encargada de iniciar la transformación de las estructuras, por tal razón fue necesario incluir en este punto, una referencia a la función transformEquikeepClause.

```
typedef struct Query
{
        NodeTag     type;
        CmdType     commandType;
        ...
        List        *rtable;        /* list of range table entries */
        FromExpr    *jointree;      /* table join tree (FROM and WHERE *
                                       clauses) */
        List        *rowMarks;      /* integer list of RT indexes of relations
                                       that are selected FOR UPDATE */
        List    *targetList;  /* target list (of TargetEntry) */
+       List    *equikeepClause;/* una lista para las expresiones
                                          lógicas de equikeepClause */
        List    *groupClause;     /* a list of GroupClause's */
        Node    *havingQual       /* qualifications applied to groups */
        List    *distinctClause; /* a list of SortClause's */
        List    *sortClause;      /* a list of SortClause's */
        ...
} Query;
```

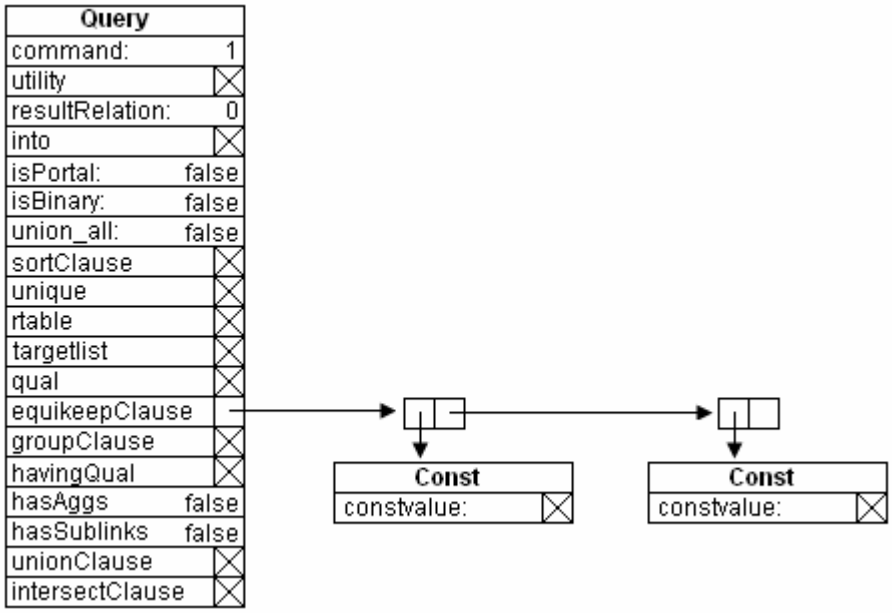

Figura 40. Query modificado

5.3.2 Planner/Optimizer.

Para adicionar el operador *Equikeep* al plan de ejecución se realizaron las siguientes modificaciones:

- Con el fin de poder reconocer el nuevo operador en esta etapa, primero se asigna una etiqueta (*T_Equikeep*) en la sección PLAN NODES del archivo *…/src/include/nodes/nodes.h* y segundo, se define la estructura de datos para este operador en el archivo *…/src/include/nodes/plannodes.h.* La figura 41 presenta el nuevo nodo para *Equikeep*.

```
+       typedef struct EquiKeep
+       {
+           Plan            plan;
+           EquiKeepState   *kstate;
+           List        *kqual;
+       } EquiKeep;
```

Figura 41. Nodo Equikeep

- Para identificar la nueva estructura *EquikeepState* primero, se adiciona la etiqueta *T_EquikeepState* en el archivo *…/src/include/nodes/nodes.h* y luego se define su estructura en el archivo *src/include/nodes/execnodes.h* como lo muestra la figura 42. *EquikeepState* es la encargada de llevar la información necesaria para retornar o almacenar las tuplas a través de las funciones de interfaz de nodo cuando el *Executor* lo requiera

```
+       typedef struct EquiKeepState
+       {
+           CommonScanState csstate;
+           List        *kqual;
+       } EquiKeepState;
```

Figura 42. EquikeepState

Los campos presentes en la estructura *EquikeepState* son:

- *csstate* (CommonScanState): Representa la información básica del estado en que se encuentra el nodo.
- **kqual* (List): Lista que almacena las expresiones lógicas para cada atributo de la relación.

- Para crear el nuevo nodo Equikeep con los datos almacenados en el *Query* y los valores actuales del costo de ejecución, se define la función *make_Equikeep* en el archivo *…/src/include/optimizer/planmain.h* y se implementa físicamente en el archivo *…/src/backend/optimizer/plan/createplan.c* (figura 43).

- Antes de unir el nuevo nodo al plan de ejecución, se necesitó transformar todas las expresiones lógicas del *Equikeep* al formato manejado por el *Executor* (Forma Normal Conjuntiva CNF, Forma Normal Disyuntiva DNF). Para lograr esas transformaciones fue necesario modificar la función *Subquery_planner* del archivo *…/src/backend/optimizar/plan/planner.c* como lo muestra la figura 44.

```
+       EquiKeep *
+       make_equikeep(List *kqual,Plan *lefttree)
+       {
+               EquiKeep   *node = makeNode(EquiKeep);
+               Plan       *plan = &node->plan;
+
+               /* cost should be inserted by caller */
+               copy_plan_costsize(plan, lefttree);
+               plan->state = (EState *) NULL;
+               plan->targetlist = lefttree->targetlist;
+               plan->qual = kqual;
+               plan->lefttree = lefttree;
+               plan->righttree = NULL;
+               node->kqual = kqual;
+               node->kstate = (EquiKeepState *) NULL;
+
+               return node;
+       }
```

Figura 43. Función make_equikeep

```
        Subquery_planner(Query *parse, double tuple_fraction) {
                ...
                List        *newkeep;
+               List        *klst;
+               ...
+               newkeep = NIL;
+               foreach(klst, parse->keepQual)
+               {
+                       Node  *transkqual;
+                       Node  *kqual = (Node *) lfirst(klst);
+
+                       transkqual = preprocess_expression(parse,
+                                       kqual,EXPRKIND_WHERE);
+
+                       newkeep = lappend(newkeep, transkqual);
+               }
+               parse->keepQual = newkeep;
+
                ...
        }
```

Figura 44. Función Subquery_planner modificado

- La función grouping_planner del archivo …*/src/backend/optimizar/plan/planner.c* es la encargada de asignar el orden de los nodos en el plan de ejecución, por tal razón es necesario incluir en este punto, una referencia a la función *make_equikeep*, como lo muestra la figura 45.

```
                static Plan *
                grouping_planner(Query *parse, double tuple_fraction)
                {
                …
+               if (parse->keepQual){
+                       result_plan =  (Plan *) make_equikeep((List *)
+                                       parse->keepQual, result_plan);
+               }
                …
                }
```

Figura 45. Función grouping_planner modificado

- Por ultimo en la función set_plan_references del archivo *…/src/backend/optimizer/plan/setrefs.c* se incluye la etiqueta *T_Equikeep* y se llama la función *fix_expr_references* para que se complete el plan de ejecución y se realicen los ajustes necesarios para que el Executor pueda trabajar adecuadamente. La figura 46 presenta el plan de ejecución de la primitiva *Equikeep On* en un operador complejo.

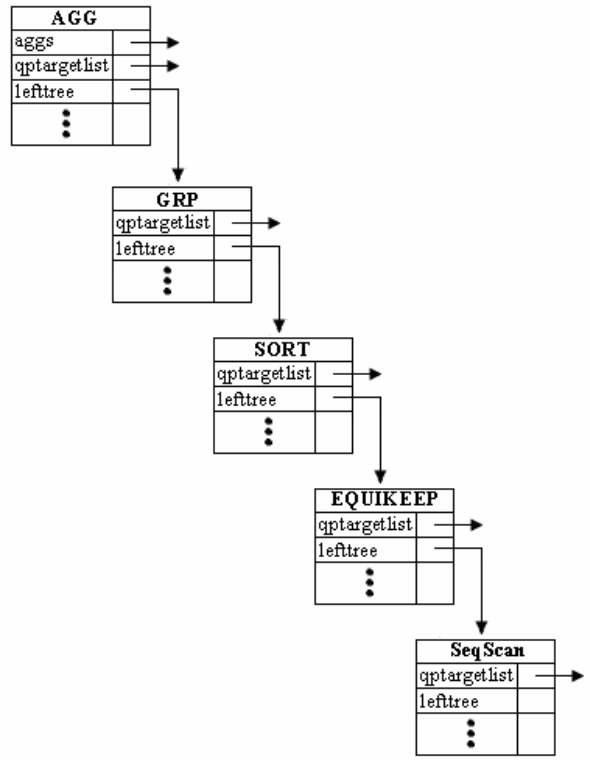

Figura 46. Plan de Ejecución de Equikeep On

5.3.3 Executor.

Al igual que *Associator*, se necesita para *Equikeep* definir e implementar las rutinas de manipulación del nodo (ExecInitEquikeep, ExecEquikeep y ExecEndEquikeep), para lo cual se realiza las siguientes modificaciones:

Crear los archivos *…/src/include/executor/nodeEquikeep.h* y *…/src/backend/executor/nodeEquikeep.c* en los cuales se define e implementa las funciones de manipulación. Estas funciones se muestran en la figura 47.

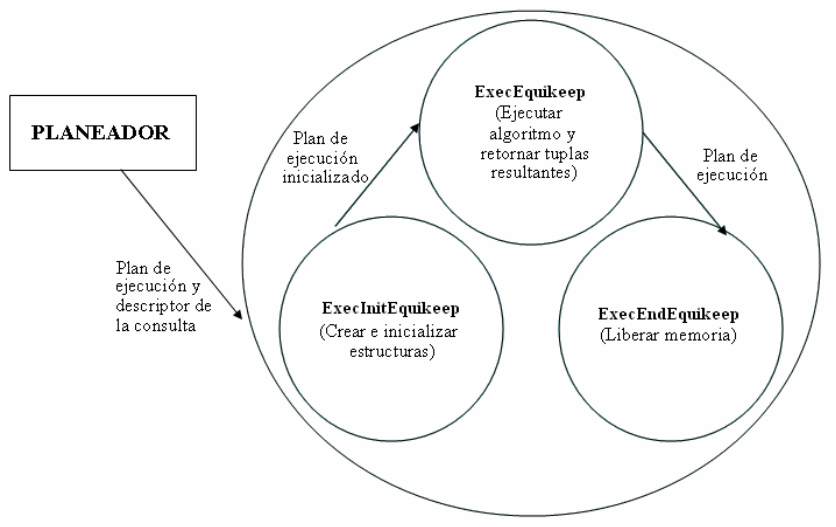

Figura 47. Funciones de manipulación del nodo Equikeep

- Con el fin de inicializar el nodo y sus estructuras cuando éste se invoque, se necesita agregar la etiqueta *T_Equikeep* en la evaluación que hace la función *ExecInitNode* del archivo *.../src/backend/executor/execProcNode.c* (figura 48).

```
bool
        ExecInitNode(Plan *node, EState *estate, Plan *parent)
        {
                ...
                switch (nodeTag(node))
                {
                ...
+                       case T_EquiKeep:
+                               result = ExecInitEquiKeep((EquiKeep *) node,
+                                                                       estate, parent);
+                       break;
                ...
        }
}
```

Figura 48. Función ExecInitNode modificado

- La función *ExecProcNode* del archivo *.../src/backend/executor/execProcNode.c* es la encargada de identificar e inicializar la ejecución de los nodos incluidos en el *QueryPlan*. Por tal razón, se modifica esta función para que pueda identificar al nodo del operador *Equikeep* (figura 49).

```
TupleTableSlot *
ExecProcNode(Plan *node, Plan *parent)
{
        …
        switch (nodeTag(node))
        {
         …
+               case T_EquiKeep:
+                       result = ExecEquiKeep((EquiKeep *)
+                                                               node);
+                       break;
        …
        }
}
```

Figura 49. Función ExecProcNode modificado

- Se agregó el identificador del nodo *Equikeep* en la función *ExecEndNode* del archivo *.../src/backend/executor/execProcNode.c*, con el fin de liberar todos los recursos reservados para la ejecución de *Equikeep* (figura 50).

```
void
ExecEndNode(Plan *node, Plan *parent)
{
        …
        switch (nodeTag(node))
        {
                …
+               case T_EquiKeep:
+                       ExecEndEquiKeep((EquiKeep *) node);
+                       break;
                …
        }
}
```

Figura 50. Función ExecEndNode modificado

5.4 Implementación de la primitiva *Mate By* al interior del motor de PostgreSQL.

Para la implementación de la primitiva Mate by, se deben modificar, como en los anteriores operadores, las estructuras, funciones y crear nuevos nodos en las etapas que componen la capa intermedia de la arquitectura de Postgres:

- La etapa Parser ha sido modificada para que construya, transforme y adjunte a las estructuras del compilador una lista de atributos condición y el atributo clase.

- El Planner/Optimizer recibe el Parser tree, verifica la lista de éste operador y agrega un nodo *Mate* al *Queryplan.*

- El Executor se ha modificado para generar por cada una de las tuplas, todas los posibles combinaciones formadas por los valores no nulos de los atributos pertenecientes a la lista de *Atributos Condición* y el valor no nulo del atributo denominado *Atributo Clase*.

5.4.1 Parser.

Al igual que en los operadores anteriores, no modifica el análisis léxico ya que no se agregan nuevos símbolos. Para el análisis sintáctico se introducen cambios y nuevo código en las funciones o estructuras de los siguientes archivos:

- Con el fin de introducir las nuevas palabras reservadas *Mate By* se modifica el archivo *…/src/backend/parser/keywords.c* donde las palabras reservadas se listan en la estructura *ScanKeywords*, manteniendo un estricto orden alfabético.

- En el archivo *…/src/backend/parser/gram.y* se adiciona las nuevas reglas de producción siguiendo la estructura lógica de este archivo especial de gramática para hacer funcional las nuevas producciones sin alterar las existentes, tal como muestra la figura 50.

```
+               mate_clause:
+                       MATE_P BY sortby_list WITH sortby
+                                                       { $$ = $3; $$ = lappend($3, $5);}
+                       | /*EMPTY*/                     { $$ = NIL; }
+               ;
```

Figura 50. Nuevas reglas de Producción

Ahora, en la regla *simple_select* se establece el punto para activar las nuevas reglas de producción (figura 51).

- Para recibir y almacenar los atributos condición y el atributo clase del operador Mate, se agregó una lista a la estructura *SelectStmnt (…/src/include/nodes/parsenodes.h*). La figura 52 presenta la estructura *SelectStmnt* modificada.

```
simple_select:
```

```
                SELECT opt_distinct target_list
                into_clause from_clause where_clause
!               mate_clause group_clause having_clause
                        {
                                SelectStmt *n = makeNode(SelectStmt);
                                n->distinctClause = $2;
                                n->targetList = $3;
                                n->into = $4;
                                n->intoColNames = NIL;
                                n->fromClause = $5;
                                n->whereClause = $6;
+                               n->mateClause = $7;
                                n->groupClause = $8;
                                n->havingClause = $9;
                                $$ = (Node *)n;
                        }
```

Figura 51. Regla simple_select modificada.

```
        typedef struct SelectStmt
        {       NodeTag         type;

                List    *distinctClause;
                RangeVar  *into;          /* target table (for select into table) */
                List    *intoColNames;    /* column names for into table */
                List    *targetList;      /* the target list (of ResTarget) */
                List    *fromClause;              /* the FROM clause */
                Node    *whereClause;             /* WHERE qualification */
+               List    *mateClause;              /* lista de mate*/
                List    *groupClause;             /* GROUP BY clauses */
                Node    *havingClause;    /* HAVING conditional-expression */
                ...
        } SelectStmt;
```

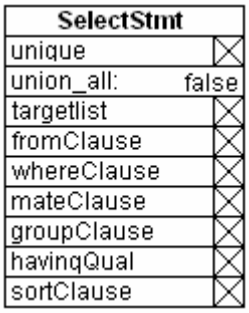

Figura 52. SelectStmnt modificada

Para continuar normalmente la etapa de transformación (transformar el Parser tree a un nodo Query) fue necesario realizar las siguientes modificaciones:

- Para llevar los datos almacenados de la estructura *SelectStmnt* a un nodo Query, se creo la función *transformMateClause*, cuya definición esta en el archivo *…/src/include/parser/parser_clause.h* y se implementó físicamente en *…/src/backend/parser/parser_clause.c* (figura 53).

```
+       List *
+       transformMateClause(ParseState *pstate, List *matelist, List *targetlist)
+       {
+               List        *mlist = NIL;
+               List        *olitem;
+
+               foreach(olitem, matelist)
+               {
+                       SortGroupBy *sortby = lfirst(olitem);
+                       TargetEntry *tle;
+                       tle = findTargetlistEntry(pstate, sortby->node,targetlist,
+                                                       ORDER_CLAUSE);
+                       mlist = lappend(mlist, sortby);
+               }
+
+               return matelist;
+       }
```

Figura 53. Función transformMateClause

- Se modificó el nodo Query (…/src/include/nodes/parsenodes.h) con el fin de que pueda recibir y almacenar los datos de la estructura SelectStmnt que pertenecen al nuevo operador. La figura 54 presenta el nodo Query modificado.

```
        typedef struct Query
        {
                NodeTag     type;
                CmdType     commandType;
                ...
                List        *rtable;        /* list of range table entries */
                FromExpr    *jointree;      /* table join tree (FROM and WHERE *
                                               clauses) */
                List        *rowMarks;      /* integer list of RT indexes of relations
                                               that are selected FOR UPDATE */
                List        *targetList;  /* target list (of TargetEntry) */
+               List        *mateClause;    /* una lista para los atributos condición
                                                   y el atributo clase */
                List        *groupClause;   /* a list of GroupClause's */
                Node        *havingQual     /* qualifications applied to groups */
                List        *distinctClause; /* a list of SortClause's */
```

```
        List    *sortClause;    /* a list of SortClause's */
        ...
} Query;
```

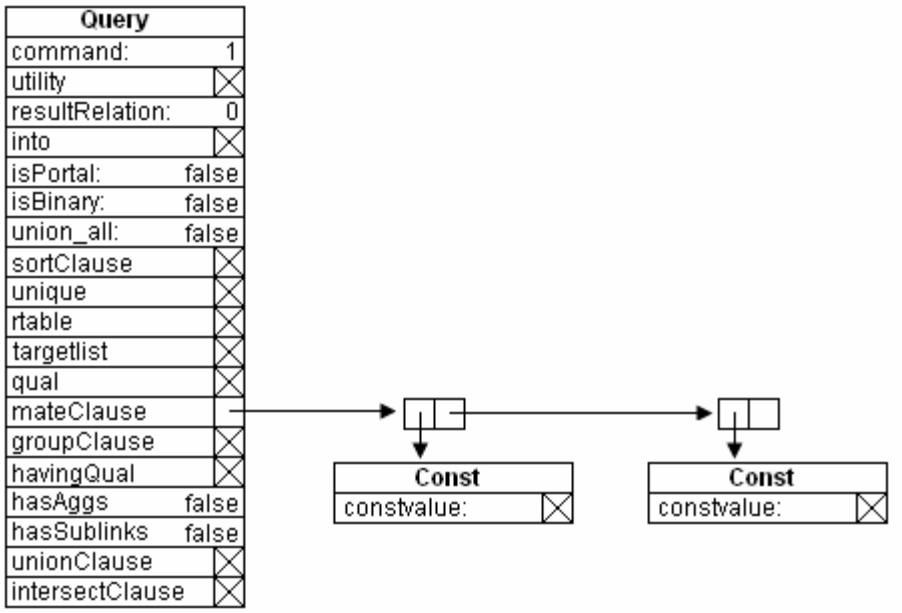

Figura 54. Query modificado

La función *transformSelectStmt* del archivo *…/src/backend/parser/analize.c* es la encargada de iniciar la transformación de las estructuras, por tal razón fue necesario incluir una referencia a la función *transformMateClause*.

5.4.2 Planner/Optimizer.

Para adicionar el operador Mate al plan de ejecución fue necesario realizar las siguientes modificaciones:

- Con el fin de poder reconocer el nuevo operador en esta etapa, primero se asigna una etiqueta (*T_Mate*) en la sección PLAN NODES del archivo *…/src/include/nodes/nodes.h* y segundo, se define la estructura de datos para este operador en el archivo *…/src/include/nodes/plannodes.h.* La figura 55 presenta el nuevo nodo para Mate.

```
+       typedef struct Mate
+       {
+               Plan        plan;
+               List        *mateList;
+               MateState   *matestate;
+       } Mate;
```

Figura 55. Nodo Mate

- Para identificar la nueva estructura MateState primero se adiciona la etiqueta *T_MateState* en el archivo *…/src/include/nodes/nodes.h* y segundo se define su estructura en el archivo *src/include/nodes/execnodes.h* como lo muestra la figura 56. *MateState* es la encargada de llevar la información necesaria para retornar o almacenar las tuplas a través de las funciones de interfaz de nodo cuando el Executor lo requiera.

```
+       typedef struct MateState
+       {
+               CommonScanState csstate;
+               List            *mateList;
+               bool            pedir_tupla;
+               bool            retornar_tupla;
+               int             num_tuplas;
+               int             out_tuplas;
+               TupleTableSlot      *tupla;
+               void                *tuplestorestate;
+       } MateState;
```

Figura 56. MateState

Los campos presentes en la estructura *MateState* son:

- csstate (CommonScanState): Representa la información básica del estado en que se encuentra el nodo.
- pedir_tupla (bool): Utilizado para que el nodo SeqScan retorne o no una tupla.
- retornar_tupla (bool): Utilizado para que el nodo superior reciba o no las tuplas generadas.
- num_tuplas (int): Cuenta cuantas tuplas ha generado el algoritmo.
- out_tuplas (int): Cuenta cuantas tuplas de las generadas por el algoritmo han sido entregadas al nodo superior.
- rginicio, rgfin (int): Controlan el rango de inicio y de fin de las combinaciones que se deben generar.
- *tupla (TupleTableSlot): Almacena la tupla pasada por el nodo *SeqScan*.
- *tuplestorestate (tuplestorestate): Almacena las tuplas generadas por el algoritmo para después pasarlas al nodo superior.

- Para crear el nuevo nodo *Mate* con los datos almacenados en el *Query* y los valores actuales del costo de ejecución, se define la función *make_mate* en el archivo *…/src/include/optimizer/planmain.h* y se implementa físicamente en el archivo *…/src/backend/optimizer/plan/createplan.c* (figura 57).

- La función *grouping_planner* del archivo *…/src/backend/optimizar/plan/planner.c* es la encargada de asignar el orden de los nodos en el plan de ejecución, por tal razón es necesario incluir en este punto, una referencia a la función *make_mate*, como lo muestra la figura 58

```
+       Mate *
+       make_mate(List *qptlist,List *mlist,Plan *lefttree)
+       {
+               Mate    *node = makeNode(Mate);
+               Plan    *plan = &node->plan;
+
+               copy_plan_costsize(plan, lefttree);
+
+               plan->state = (EState *) NULL;
+               plan->targetlist = qptlist;
+               plan->qual = NIL;
+               plan->lefttree = lefttree;
+               plan->righttree = NULL;
+
+               node->mateList = mlist;
+               node->matestate = (MateState *) NULL;
+
+               return node;
+       }
```

Figura 57. Función make_mate

```
        static Plan *
        grouping_planner(Query *parse, double tuple_fraction)
        {
        …
+       if (parse->mateClause)     {
+               List *alist;
+
+               if (parse->hasAggs)
+                       alist = new_unsorted_tlist(result_plan->targetlist);
+               else
+                       alist = tlist;
+               result_plan = (Plan *) make_mate(alist,
+                                               parse->mateClause,result_plan);
+               }
        …
        }
```

Figura 58. Función grouping_planner modificado

- Por ultimo en la función *set_plan_references* del archivo *…/src/backend/optimizer/plan/setrefs.c* se incluye la etiqueta *T_Mate* para que se complete el plan de ejecución y se realicen los ajustes necesarios para que el *Executor* pueda trabajar adecuadamente. La figura 59 presenta el plan de ejecución de la primitiva Mate By en un operador complejo.

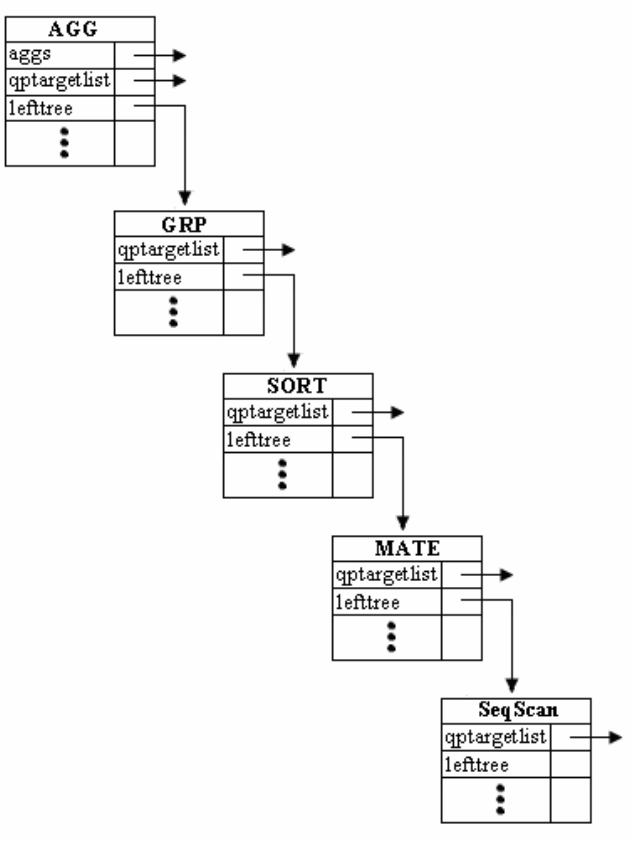

Figura 59. Plan de Ejecución de Mate By

5.4.3 Executor.

Al igual los anteriores operadores, para *Mate* se necesitan definir e implementar las rutinas de manipulación del nodo (ExecInitMate, ExecMate y ExecEndMate), para lo cual se realiza las siguientes modificaciones:

- Crear los archivos *…/src/include/executor/nodeMate.h* y *…/src/backend/executor/nodeMate.c* en los cuales se define e implementa las funciones de manipulación. Estas funciones se muestran en la figura 60.

- Con el fin de inicializar el nodo y sus estructuras cuando éste se invoque, se necesita agregar la etiqueta *T_Mate* en la evaluación que hace la función *ExecInitNode* del archivo *.../src/backend/executor/execProcNode.c* (figura 61)

- La función ExecProcNode del archivo *.../src/backend/executor/execProcNode.c* es la encargada de identificar e inicializar la ejecución de los nodos incluidos en el *QueryPlan*. Por tal razón, se modificó esta función para que pueda identificar al nodo del operador *Mate* (Figura 62).

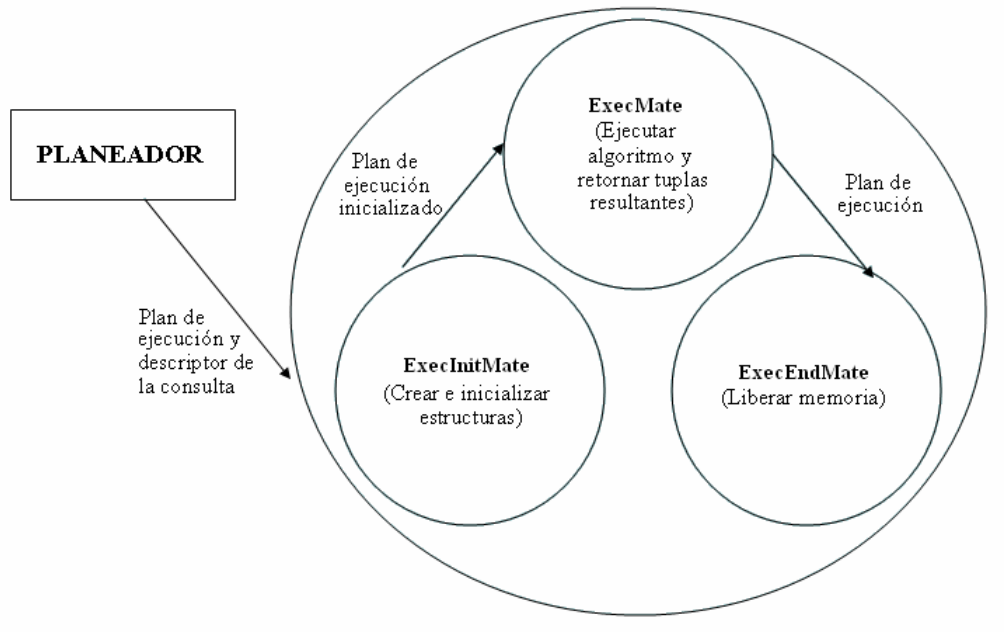

Figura 60. Funciones de manipulación del nodo Mate

```
        bool
        ExecInitNode(Plan *node, EState *estate, Plan *parent)
        {       ...
                switch (nodeTag(node))
                {...
+                       case T_Mate:
+                               result = ExecInitMate((Mate *) node,
+                                                       estate,
+                                                       parent);
+                       break;
                ...
        }
}
```

Figura 61. Función ExecInitNode modificado

```
        TupleTableSlot *
        ExecProcNode(Plan *node, Plan *parent)
        {       …
                switch (nodeTag(node))
                { ...
+                       case T_Mate:
+                                       result = ExecMate((Mate *) node);
+                       break;
                …
                }
        }
```

Figura 62. Función ExecProcNode modificado

- Se agregó el identificador del nodo *Mate* en la función *ExecEndNode* del archivo *.../src/backend/executor/execProcNode.c*, con el fin de liberar todos los recursos reservados para la ejecución de *Mate* (figura 63).

```
        void
        ExecEndNode(Plan *node, Plan *parent)
        {
                …
                switch (nodeTag(node))
                {
                        …
+                       case T_Mate:
+                               ExecEndMate((Mate *) node);
+                       break;
                        …
                }
        }
```

Figura 63. Función ExecEndNode modificado

5.5 Implementación de los operadores *Entro, Gain y Describe Classifier* como funciones definidas por usuario (FDU) en PostgreSQL

PostgreSQL le proporciona al usuario la capacidad de implementar funciones definidas por el usuario (FDU), las cuales se escriben en el lenguaje de programación anfitrión C o en un lenguaje procedural basado en SQL como *PL/pgSQL*. En este último caso, es necesario registrar el lenguaje *PL/pgSQL*, en la base de datos, donde se crearán las FDUs. Este proceso se hace accediendo como usuario *dba postgres* (o un usuario con privilegios equivalentes) y ejecutando la instrucción:

$ CREATE LANGUAJE PLPGSQL <<dbname>>,

donde *dbname* es el nombre de la base de datos.

PostgreSQL incluye entre sus directorios, el directorio *Contrib* ubicado en *./postgresql/contrib*, para que el usuario pueda definir sus funciones y establecer la jerarquía necesaria para su correcto funcionamiento.

Una vez codificada la FDU, es necesario crearla en la base de datos donde se va a utilizar. Este proceso se puede realizar, a través de un *script* SQL, donde se encuentran todas las instrucciones de la función. Una vez conectado a la base de datos, se ejecuta este *script* con la instrucción de plsql *\i /ruta_archivo/nombre_archivo* y la FDU se crea.

5.5.1 Implementación de la FDU Entro().

Entro() permite calcular la entropía de un conjunto de datos. Los archivos que contienen las fuentes de la función *entro()* se ubican en el directorio:

./postgresql_7.3.4/contrib/kdd/clasificacion/entro. El *script entro.sql* contiene las instrucciones en PL/pgSQL necesarias para la creación de *entro().*

Con la instrucción \i /postgresql_7.3.4/contrib/kdd/clasificacion/entro/entro.sql, se crea la FDU *entro()* y se obtiene el mensaje de confirmación '*CREATE FUNCTION*'.

La función *entro()* recibe como parámetro de entrada un tipo de dato VARCHAR que contiene el nombre de la tabla sobre la cual se realizaran los cálculos para obtener los valores de entropía. Dentro de la cláusula SELECT, *entro()* tiene la siguiente sintaxis:

SELECT * FROM *Entro* (`<*NombreTabla*>')

Inicialmente recorre la tabla para transformar los valores contenidos a fin de agilizar y optimizar procesos. Convierte la tabla original en una tabla transformada *'mate_entro'* conteniendo únicamente valores numéricos, además adiciona el atributo '*entro*', el cual contiene el valor de entropía calculado para la combinación en cada tupla o grupo distinto de ellas. Crea además la tabla *'mate_values'*, que guarda los parámetros de transformación utilizados, necesarios para la conversión, cuando se necesite expresar los valores resultantes finales del proceso de clasificación, en términos de los valores originales.

En general, la estructura de *'mate_entro'* puede variar en el número de atributos condición del mismo tipo numérico y contiene siempre como ultimo atributo, '*entro*'. La estructura de la tabla *'mate_values'* será siempre igual.

Ejemplo 13. Sea la tabla APARIENCIA de la figura 64, Calcular la entropía de las diferentes combinaciones de los atributos condición *tamano, cabello, ojos* con el atributo clase *cutis*.

tamano	cabello	ojos	cutis
corto	oscuro	azul	claro
alto	oscuro	cafe	oscuro
alto	rojo	azul	claro
corto	claro	azul	oscuro
alto	claro	azul	claro
alto	oscuro	azul	claro
alto	claro	cafe	oscuro
corto	oscuro	cafe	oscuro

Figura 64. Tabla de clasificación Apariencia

Mediante la siguiente instrucción SQL se obtiene las diferentes combinaciones de los atributos condición con el atributo clase, utilizando la primitiva *MATE BY*:

SELECT *tamaño,cabello,ojos,cutis,count(*)* INTO *mateapariencia*
FROM *apariencia*
MATE *BY tamano,cabello,ojos* WITH *cutis*
GROUP BY *tamano,cabello,ojos,cutis*

Mediante la instrucción:

SELECT * FROM ENTRO(`*Mateapariencia*`)

se aplica la función *entro()* sobre la tabla *Mateapariencia* y se obtiene las tablas *Mate_values*, con la codificación de los valores de los atibutos, y *Mate_entro*, con la entropia de las diferentes combinaciones de los atributos condición, con el atributo clase. Las figuras 65 y 66 muestran estas tablas.

nro	atributo	valor	discret	nclases
1	tamano	alto	0	
2	tamano	corto	1	2
3	cabello	claro	0	
4	cabello	oscuro	1	
5	cabello	rojo	2	3
6	ojos	azul	0	
7	ojos	cafe	1	2
8	cutis	claro	0	
9	cutis	oscuro	1	2

Figura 65. Tabla de clasificación Mate_values

tamano	cabello	ojos	cutis	count	entro
0	0	0	0	1	0
0	0	1	1	1	0
0	0	-1	0	1	500
0	0	-1	1	1	500
0	1	0	0	1	0
0	1	1	1	1	0
0	1	-1	0	1	500
0	1	-1	1	1	500
0	2	0	0	1	0
0	2	-1	0	1	0
0	-1	0	0	3	0
0	-1	1	1	2	0
0	-1	-1	0	3	442
0	-1	-1	1	2	529
1	0	0	1	1	0
...					
...					

Figura 66. Tabla de clasificación Mate_entro

5.5.2 Implementación de la FDU Gain().

La FDU Gain() implementa el operador *Gain* . La función Gain() calcula la ganancia de información y genera una tabla las reglas de clasificación. Los archivos que contienen las fuentes de la función *gain()* se ubican en el directorio: *./postgresql_7.3.4/contrib/kdd/clasificacion/gain* que corresponden a:

- gain.c: contiene las rutinas principales en lenguaje C necesarias para la ejecución de gain().
- arbol.c: rutinas de manipulación del árbol construido en memoria principal, requerido por el algoritmo.
- busqueda.c: rutinas para recorrer el árbol y condicionar la búsqueda de registros en la tabla pasada como parámetro.
- nodearbolatabla.c: convierte el árbol en una tabla física para hacer la información persistente y generar las reglas de clasificación.
- defs.h : algunas definiciones importantes.
- makefile: archivo de compilación necesario para subrutinas en C.
- gain.sql: contiene la instrucción de creación y definición de la función principal gain().
- gain.so: este archivo solo existe después de compilar, y contiene código binario de uso exclusivo del servidor postgres.

La compilación de la función escrita en C se debe realizar como usuario *root*, o como un usuario con permisos equivalentes. Desde el directorio *../contrib/kdd/clasificacion/gain* se ejecuta el comando *make* (o *gmake*) para compilar, y el comando *make install* (*o gmake install*) para instalar.

Hasta aquí se ha creado el código ejecutable de la función gain () y subrutinas en *C*. Para la utilización de la función, se ingresa como usuario Postgres o equivalente, a la base de datos a través de la terminal interactiva psql. Una vez dentro se ejecuta la instrucción:

\i ./postgres_7.3.4/contrib/kdd/clasificacion/gain/gain.sql

Con esta instrucción se crea la función, y obtiene el mensaje de confirmación *CREATE FUNCTION*.

La funcion *Gain()* recibe como parámetro de entrada un dato de tipo VARCHAR que contiene el nombre de la tabla, resultado de la función *entro()*, i.e. una tabla con el formato *Mate_entro*. Dentro de la cláusula SELECT, gain() tiene la siguiente sintaxis:

SELECT * FROM Gain (`<*mate_entro*>')

La función Gain(), extrae de la tabla *Mate_values* cuantos atributos posee *Mate_entro* y el número de valores posibles para cada atributo existente. Extrae los valores de entropía y calcula la ganancia de información para establecer que atributo se convierte en la raíz del árbol de decisión. A partir de aquí, recorre la tabla para buscar registros cuyos atributos

coincidan con los valores existentes en el árbol para calcular las nuevas ganancias de información; selecciona el atributo de mayor ganancia y se agrega en la respectiva posición, en el árbol.

Este proceso es iterativo hasta evaluar todas las combinaciones de los atributos condición con el atributo decisión (*atributo clase*).

Las subrutinas *TablaArbol()* y *TablaRules()* recorren el árbol en memoria y a partir de él construyen dos tablas, que contienen información relevante para generar las reglas de clasificación a partir de los nodos existentes, o pseudo reglas que es necesario convertir a los valores originales previos a la transformación. Estas tablas son por defecto *Tclases* que contiene los nodos, y *Trulesclases* que contiene las pseudo reglas, cuyas estrucutras se muestran en las figuras 67 y 68 respectivamente.

NODO	PADRE	ATRIBUTO	VALOR	CLASE
0	-1	2	-1	-1
1	0	2	0	-1
2	1	0	0	0
3	1	0	1	-1
4	3	1	0	1
5	3	1	1	0
6	3	1	2	-999
7	0	2	1	1

Figura 67. Tabla Tclases

La tabla *Tclases* especifica las relaciones entre los nodos del árbol, y los valores útiles. Los valores están transformados, y por tanto pueden traducirse en resultados reales y coherentes. En este caso el valor constante ‘-999‘ indica un caso especial en que por el valor de ganancia el atributo clase pude ser cualquiera de los posibles, y el valor -1 indica que el valor del atributo aún no es nodo terminal.

La tabla *Trulesclases* presenta los nodos relacionados en una misma regla de forma secuencial. Todos los registros que conforman una regla tienen el mismo valor en el atributo *id*, que indica el número de la regla. El valor constante ‘-777’ se sustituye por el nombre del atributo clase para representar el consecuente de una regla, y el valor constante ‘-999’ indica la misma excepción utilizada para la tabla *Tclases*.

Por ejemplo, en la tabla *Trulesclases* de la figura 68, la primera regla se interpreta como:

Si atributo[0] = valor[0] y atributo[2] = valor[0] entonces atributo[clase] = valor[0]

que significa:

Si *tamaño* = *alto* y *ojos* = *azul* entonces *cutis* = *claro*.

que es una regla de clasificación válida para la tabla original de la figura 64

id	atributo	valor
1	0	0
1	2	0
1	-777	0
2	1	0
2	0	1
2	2	0
2	-777	1
3	1	1
3	0	1
3	2	0
3	-777	0
4	1	2
4	0	1
4	2	0
4	-777	-999
5	2	1
5	-777	1

Figura 68. Tabla Trulesclases

5.6 Implementación del operador Describe Associator como la FDU Describe_Association_Rules en PostgreSQL

Describe_association_rules permite generar las reglas de Asociación. Los archivos que contienen las fuentes de esta función se ubican en el directorio: *./postgresql_7.3.4/contrib/kdd/asociacion* que corresponden a:

- type_rules.sql: define el tipo de dato de retorno necesario para el resultado de la función describe_association_rules().
- size_rules.c: calcula el tamaño de atributos no nulos de la tabla de asociación definida por el usuario.
- assoc_rules.sql: genera las reglas candidatas de una tabla especifica con un tamaño de regla definido por el usuario.
- describe_rules.sql: genera las reglas finales de acuerdo a la confianza definida por el usuario en el inicio de la función.
- makefile: archivo de compilación necesario para subrutinas en C.
- size_rules.sql: contiene la instrucción de creación y la definición de la función size_rules().
- size_rules.so: este archivo solo existe después de compilar, y contiene código binario de uso exclusivo del servidor postgres.

Para la función *size_rules*, escrita en lenguaje *C*, la compilación se debe realizar como usuario *root*, o como un usuario con permisos equivalentes. Desde el directorio *../contrib/kdd/asociacion* se ejecuta el comando *make* (o *gmake*) para compilar, y el comando *make install* (o *gmake install*) para instalar; obteniendo el código ejecutable de la función *size_rules*().

Para la utilización de la función *describe_association_rules*(), se ingresa como usuario Postgres o equivalente, a la base de datos a través de la terminal interactiva psql. Una vez dentro se ejecutan las instrucciones en el siguiente orden:

\i ./postgres../contrib/kdd/asociacion/size_rules.sql
\i ./postgres../contrib/kdd/asociacion/type_rules.sql
\i ./postgres../contrib/kdd/asociacion/assoc_rules.sql
\i ./postgres../contrib/kdd/asociacion/describe_rules.sql

Recibiendo el mensaje de confirmación 'CREATE FUNCTION' por cada una de las instrucciones anteriores. A partir de este momento ya es posible invocar a la función *describe_association_rules().*

Describe_association_rules(), toma como parámetros de entrada, un dato de tipo VARCHAR que contiene el nombre de la tabla, generada por la primitiva *Associator Range* descrita anteriormente, un dato de tipo INTEGER mayor de 1 especificando el tamaño de las reglas deseadas y un dato de tipo NUMERIC entre 0 y 100 que especifica el porcentaje de confianza mínima de las reglas a generar. Dentro de la cláusula SELECT, *describe_associaion_rules()* tiene la siguiente sintaxis:

SELECT * FROM
DESCRIBE_ASSOCIATION_RULES(`*<NombreTabla>*',*<TamañoRegla>*,*<Confianza>*)

Ejemplo 14. Sea la tabla Transaccion de la figura 69. Generar las reglas de Asociación de tamaño 4 con un soporte mínimo de 1 y una confianza del 80%.

a	b	c	d
a1	b1	c1	d1
a1	b2	c1	d2

Figura 69. Tabla transacción

Con la siguiente cláusula SQL se genera los itemsets frecuentes:

SELECT *a,b,c,d,count(*)* AS *soporte* INTO *assotransaccion*
FROM *transaccion*
ASSOCIATOR RANGE 1 UNTIL 4
GROUP BY *a,b,c,d* HAVING *count(*)*>=1

El resultado de *Associator Range* se muestra en figura 70.

a	b	c	d	soporte
a1	b1	c1	d1	1
a1	b1	c1		1
a1	b1		d1	1
a1	b1			1
a1	b2	c1	d2	1
a1	b2	c1		1
a1	b2		d2	1
a1	b2			1
a1		c1	d1	1
a1		c1	d2	1
a1		c1		2
a1			d1	1
a1			d2	1
a1				2
	b1	c1	d1	1
	b1	c1		1
	b1		d1	1
	b1			1
	b2	c1	d2	1
	b2	c1		1
	b2		d2	1
	b2			1
		c1	d1	1
		c1	d2	1
		c1		2
			d1	1
			d2	1

Figura 70. Tabla Assotransaccion

Con la siguiente cláusula SQL se generan las reglas de asociación de tamaño 4 con una confianza mínima de 80%.

SELECT * FROM DESCRIBE_ASSOCIATION_RULES(`AssoTransacción',4,80)

En la Figura 71 se muestra el resultado de aplicar la función *describe_association_rules()* en *Assotransaccion*.

Las reglas se representans en forma de implicación, dónde un registro representa el antecedente (atributo implica = "A") y el inmediatamente siguiente su consecuente (atributo implica = "C"). La confianza se presenta una sola vez por cada regla en el atributo confianza del registro antecedente y en el atributo *n_regla* se muestra el número de la regla generada que cumplió los parámetros ingresados por el usuario.

Algunos ejemplos de reglas según la figura 71 son:

Regla 2: **Si** *b = b1* **entonces** *a = a1* y *c = c1* y *d = d1*.
Regla 19: **Si** *a = a1* y *b = b2* **entonces** *c = c1* y *d = d2*.

a	b	c	d	n_regla	implica	soporte	confianza
-	bl	-	-	2	A	1	100.00
al	-	cl	dl	2	C	1	
-	-	-	dl	4	A	1	100.00
al	bl	cl	-	4	C	1	
al	bl	-	-	5	A	1	100.00
-	-	cl	dl	5	C	1	
al	-	-	dl	7	A	1	100.00
-	bl	cl	-	7	C	1	
-	bl	cl	-	8	A	1	100.00
al	-	-	dl	8	C	1	
-	bl	-	dl	9	A	1	100.00
al	-	cl	-	9	C	2	
-	-	cl	dl	10	A	1	100.00
al	bl	-	-	10	C	1	
al	bl	cl	-	11	A	1	100.00
-	-	-	dl	11	C	1	
al	bl	-	dl	12	A	1	100.00
-	-	cl	-	12	C	2	
al	-	cl	dl	13	A	1	100.00
-	bl	-	-	13	C	1	
-	bl	cl	dl	14	A	1	100.00
al	-	-	-	14	C	2	
-	b2	-	-	16	A	1	100.00
al	-	cl	d2	16	C	1	
-	-	-	d2	18	A	1	100.00
al	b2	cl	-	18	C	1	
al	b2	-	-	19	A	1	100.00
-	-	cl	d2	19	C	1	
al	-	-	d2	21	A	1	100.00
-	b2	cl	-	21	C	1	
-	b2	cl	-	22	A	1	100.00
al	-	-	d2	22	C	1	
-	b2	-	d2	23	A	1	100.00
al	-	cl	-	23	C	2	
-	-	cl	d2	24	A	1	100.00
al	b2	-	-	24	C	1	
al	b2	cl	-	25	A	1	100.00
-	-	-	d2	25	C	1	
al	b2	-	d2	26	A	1	100.00
-	-	cl	-	26	C	2	
al	-	cl	d2	27	A	1	100.00
-	b2	-	-	27	C	1	
-	b2	cl	d2	28	A	1	100.00
al	-	-	-	28	C	2	

Figura 71. Tabla describe_association_rules

5.7 Implementación del operador Describe Classifier como la FDU Describe_Classification_Rules() en PostgreSQL

Describe_classification_rules() permite mostrar las reglas de Clasificación generadas por la FDU Gain(). Los archivos que contienen las fuentes de la función *Describe_classification_rules() se* ubican en el directorio: *./postgresql_7.3.4/contrib/kdd/clasificacion/* que corresponden a:

- describeCrules.sql: crea la función describeCrules.
- describeCrules.c: crea la tabla de reglas.
- describe_class_rules.sql: crea el tipo rulesclass y la función *describe_classification_rules()*, la cual muestra las reglas a partir de la tabla trulesclasses generada por la FDU Gain().
- makefile: archivo de compilación necesario para subrutinas en C.

Para la función *describeCrules.c*, escrita en lenguaje *C*, la compilación se debe realizar como usuario *root*, o como un usuario con permisos equivalentes. Desde el directorio *../contrib/kdd/clasificacion* se ejecuta el comando *make* (o *gmake*) para compilar, y el comando *make install* (o *gmake install*) para instalar; obteniendo el código ejecutable de la función *describeCrules.c*.

Para la utilización de la función *describe_classification_rules()*, se ingresa como usuario Postgres o equivalente, a la base de datos a través de la terminal interactiva psql. Una vez dentro se ejecutan las instrucciones en el siguiente orden:

\i ./postgres../contrib/kdd/clasificacion/describeCrules.sql
\i ./postgres../contrib/kdd/clasificacion/describe_class_rules.sql

Recibiendo el mensaje de confirmación 'CREATE FUNCTION' por cada una de las instrucciones anteriores. A partir de este momento ya es posible invocar a la función *describe_classification_rules().* Dentro de la cláusula SELECT, *describe_classification_rules()*tiene la siguiente sintaxis:

```
SELECT * FROM describe_classification_rules();
```

6 Proceso de instalación de PostgresKDD

Esta sección describe la instalación del código fuente de PostgresKDD con las primitivas y funciones de minería de datos implementadas en PostgreSQL-7.3.4.

6.1 Requisitos del Sistema

Para poder instalar PostgresKDD, asegurese de que cumpla los siguientes requsitos:

- Un computador Pentium IV mínimo de 1.8 Ghz
- Memoria RAM mínimo de 512 Mb
- Disco duro mínimo de 40GB
- Sistema operativo Linux

6.2 Instalación y Configuración de PostgresKDD

Ingrese al sistema como usuario *root*.

- *#*

Inserte el CD de PostgresKDD en la unidad de CD-ROM y monte la unidad

- *# mount /mnt/cdrom*

Copie desde el CD, el archivo *postgresKDD.tar.gz* de la carpeta *Software* a la carpeta */usr/local/src* de su sistema

- *# cp /mnt/cdrom/Software/ postgresKDD.tar.gz /usr/local/src*

Vaya al directorio */usr/local/src* y descomprime y desempaqueta los fuentes

- *cd /usr/local/src*
- *# gunzip postgresKDD.tar.gz*
- *# tar –xvf postgresKDD.tar*

Cree un enlace al directorio *postgresql-7.3.4*

- *# ln –s postgresql-7.3.4 postgresKDD*

Vaya al directorio postgresKDD y prepare las fuentes para compilarlas:

- *# cd postgresKDD*
- *# ./configure*

Compile las fuentes de PostgresKDD

- *# make*

Instale el binario de PostgresKDD

- *# make install*

Ahora realice los siguientes pasos de configuración Post-Instalación de *PostgresKDD*:

Como root cree el grupo *postgres*

- *# groupadd postgres*

En este grupo cree al usuario *postgres* con su directorio HOME en */usr/local/pgsql*

- *# useradd -g postgres –D /usr/local/pgsql postgres*

Vaya al directorio HOME del usuario *postgres* y cree el directorio *data*

- *# cd /usr/local/pgsql*
- *# mkdir /usr/local/pgsql/data*

Cambie a propietario *postgres* y grupo *postgres* al directorio */usr/local/pgsql* y sus subdirectorios

- *# chown –R postgres:postgres /usr/local/pgsql*

Cambiese a usuario postgres e inicialice la base de datos

- *# su - postgres*
- $ */usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data*

Levante el servico Postgres

- *$ /usr/local/pgsql/bin/postmaster -i -S -D /usr/local/pgsql/data&*

Cree la base de datos a utilizar y conectese a ella

- *$ /usr/local/pgsql/bin/createdb <base_de_datos>*
- *$ /usr/local/pgsql/bin/psql <base_de_datos>*

6.3 Instalación y Configuración de las FDUs para Asociación y Clasificación

Antes de iniciar la utilización de las funciones definidas por el usuario de asociación y clasificación, es necesario instalarlas y configurarlas en la base de datos creada:

6.3.1 FDUs para Asociación

Ingrese como usuario root y se desplaza al directorio de asociación.

- *# cd /usr/local/src/postgresKDD/contrib/kdd/asociación*

Compile e instale las FDUs de asociación.

- *# make*
- *# make install*

Ingrese como usuario postgres y conectese a la base de datos que creó.

- *# su – postgres*
- *$ psql <base_de_datos>*
-

Ejecute el script *inicio_tipo.sql* que carga la función para crear el tipo de datos que utiliza la función *describe_association_rules.*

- *#> \i /usr/local/src/postgresKDD/contrib/kdd/asociación/inicio_tipo.sql*

Ejecute la sentencia SQL para determinar el tipo de la tabla de asociación

- *#> SELECT type_rules('nombre_tabla_asociacion');*

Por ejemplo

- *#>SELECT type_rules(‘transacciones’);*

Ejecute el script *inicio_funcion.sql* que carga las funciones de generación de reglas de asociación.

- *#> \i /usr/local/src/postgresKDD/contrib/kdd/asociación/inicio_funcion.sql*

Cuando haya generado los itemsets frecuentes, ejecute la sentencia SQL para generar las reglas de asociación.

- *#> SELECT * FROM describe_association_rules('nombre_tabla_asociacion', tamaño_regla,confianza);*

Por ejemplo:

- #>SELECT item1,item2,item3,item4, count(*) INTO frecuentes
  > FROM transacciones
  >ASSOCIATOR RANGE 1 UNTIL 4
  > GROUP BY item1,item2,item3,item4 HAVING count(*)> 2;
- *#> SELECT * FROM describe_association_rules('frecuentes', 3,10);*

6.3.2 FDUs para Clasificación

Ingrese como usuario root y se desplaza al directorio de clasificación

- *# cd /usr/local/src/postgresql-7.3.4/contrib/kdd/clasificacion*

Compile e instale las FDUs de Clasificación.

- *# make*
- *# make install*

Ingrese como usuario postgres y conectese a la base de datos <base_de_datos>

- *# su – postgres*
- *$ psql <base_de_datos>*

Ejecute el script *inicio.sql* que carga las funciones de clasificación.

- *#> \i /usr/local/src/postgresql-7.3.4/contrib/kdd/clasificacion/inicio.sql*

Para obtener la entropía de la tabla de clasificación, ejecutar la sentencia SQL

- *#> SELECT * FROM entro('nombre_tabla_clasificacion');*

Para obtener la tabla con el árbol de decisión para la tabla de clasificación, ejecutar la sentencia SQL

- *#> SELECT * FROM gain('mate_entro');*

Para generar las reglas de clasificación, ejecutar la sentencia SQL

- *#> SELECT * FROM describe_classification_rules();*

Por ejemplo:

- #> SELECT estado, temperatura,humedad,viento,jugar_tenis,count(*)
  >INTO clasificacion
  >FROM juegos
  >MATE BY estado, temperatura,humedad,viento WITH jugar_tenis
  >GROUP BY estado, temperatura,humedad,viento,jugar_tenis;

- #>SELECT * FROM *entro* (‘clasificacion´);
- #>SELECT * FROM *gain* (‘mate_entro’);
- *#> SELECT * FROM describe_classification_rules();*

Referencias Bibliográficas

AGRAWAL R., IMIELINSKI T., SWAMI A.(1993), Mining Association Rules between Sets of Items in Large Databases, ACM SIGMOD, Washington DC, USA.

AGRAWAL R., MEHTA M., SAFER J., SRIKANT R. (1996), The Quest Data Mining System, 2º Conference KDD y Data Mining, Portland, Oregon, USA.

AGRAWAL R., SHIM K.(1996), Developing Tightly-Coupled Data Mining Applications on a Relational Database System, The Second International Conference on Knowledge Discovery and Data Mining, Portland, Oregon.

AGRAWAL R., SRIKANT R.(1994), Fast Algorithms for Mining Association Rules, VLDB Conference, Santiago, Chile, 1994.

BRIN S., MOTWANI R., ULLMAN J., TSUR S. (1997), Dynamic Itemset Counting and Implication Rules for Market Basket Data, ACM SIGMOD, USA.

CLEAR, J., DUNN, D., HARVEY, B., HEYTENS, M., LOHMAN, P., MEHTA, A., MELTON, M., ROHRBERG, L., SAVASERE, A., WEHRMEISTER, R., XU, M.(1999), NonStop SQL/MX Primitives for Knowledge Discovery, KDD-99, San Diego, USA.

CODD, E., F. (1970), A Relational Model of Data for Large Shared Data Banks, CACM, Vol. 13, No. 6, June.

CHAUDHURI S.(1998), Data Mining and Database Systems: Where is the Intersection?, Bulletin of the Technical Committee on Data Engineering, Vol. 21 No. 1.

CHEN M., HAN J., YU P.(1996), Data Mining: An Overview from Database Perspective, IEEE Transactions on Knowledge and Data Engineering.

FAYYAD U., PIATETSKY-SHAPIRO G. y SMYTH P. (1996), Knowledge Discovery and Data Mining: Towards a Unifying Framework. In *The Second International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon.

FREITAS, A.A., LAVINGTON, S.H.(1997), Using SQL Primitives and Parallel DBServers to Speed Up Knowledge Discovery in large relational databases, University of Essex, UK, http:// citeseer.nj.nec.com/cs, 1997.

HAN J., FU Y., WANG W., CHIANG J., KOPERSKI K., LI D., LU Y., RAJAN A., STEFANOVIC N., XIA B., ZAIANE O. (1996a), DBMiner: A System for Mining Knowledge in Large Relational Databases, The second International Conference on Knowledge Discovery & Data Mining, Portland, Oregon.

HAN J., FU Y., WANG W., KOPERSKI K., ZAIANE O.(1996b), DMQL: A Data Mining Query Language for Relational Databases, SIGMOD 96 Workshop, On research issues on Data Mining and Knowledge Discovery DMKD 96, Montreal, Canada.

HAN, J., PEI J., YIN Y.(2000), Mining Frequent Patterns without Candidate Generation, ACM SIGMOD, Dallas, Texas, USA.

HAN, J., KAMBER, M.(2001), Data Mining:Concepts and Techniques, Morgan Kaufmann Publishers, San Francisco.

IMIELNSKI T., MANNILA, H.(1996), A Database Perspective on Knowledge Discovery, Communications of the ACM, Vol 39, No. 11.

IMIELINSKI T., VIRMANI A.(1999), MSQL: A Query Language for Database Mining, in journal Data Mining and Knowledge Discovery, Kluwer Academica Publishers, Volume 3, Number 4.

MEO R., PSAILA G., CERI S.(1996), A New SQL-like Operator for Mining Association Rules, VLDB Conference, Bombay, India.

MEO R., PSAILA G., CERI S.(1998), An Extension to SQL for Mining Association Rules, Data Mining and Knowledge Discovery, Kluwer Academic Publishers, Vol 2, pp .195-224, Boston.

MELTON, J., EISENBERG, A.(2001), SQL Multimedia and Application Packages (SQL/MM), in http://www.sigmod.org/records/issues/0112/standars.pdf.

NETZ A., CHAUDHURI S., BERNHARDT J., FAYYAD U.(2000), Integration of Data Mining and Relational Databases, Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egytpr.

PARK J.S., CHEN M., YU P.(1995), An Effective Hash-Based Algorithm for Mining Association Rules, ACM SIGMOD, San Jose, Ca USA.

QUINLAN, J.R.(1986), Induction of decision trees. Machine Learning, Morgan Kaufmann Publishers, pag. 81-106.

QUINLAN, J.R.(1993), C4.5: Programs for Machine Leraning, Morgan Kaufmann Publishers.

RAJAMANI, K., COX, A., IYER, B., CHADHA, A.(1999), Efficient Mining for Association Rules with Relational Database Systems, International Database Engineering and Aplication Symposium, p. 148-155.

SARAWAGI, S., THOMAS, S., AGRAWAL, R.(2000), Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications, Data Mining and Knowledge Discovery, Kluwer Academic Publishers, Vol 4.

SCHWENKREIS, F.(2000), New working draft of SQL/MM Part 6: Data Mining based on BHX008 and BHX033-BHX039, ISO/IEC JTC1/SC32 Data Management and Interchange WG$ SQL/MM, Secretariat USA (ANSI).

SATTLER, K., DUNEMANN, O.(2001), SQL Database primitives for Decision Tree Classifiers, CIKM, Atlanta, Georgia, USA.

STONEBRAKER, M., ROWE, L., A.(1986), The Design of POSTGRES, Proceedings of the ACM-SIGMOD Conference, Washington D.C.

TIMARÁN, R. (2001), Arquitecturas de Integración del Proceso de Descubrimiento de Conocimiento con Sistemas de Gestión de bases de datos: un Estado del Arte, en revista Ingeniería y Competitividad, ISSN 0123-3033, Universidad del Valle, Volumen 3, No. 2, Cali, Colombia

TIMARÁN, R., MILLÁN, M., MACHUCA,F.(2003), New Algebraic Operators and SQL Primitives for Mining Association Rules, in proceedings of the IASTED International Conference on Neural Networks and Computational Intelligence (NCI 2003), International Association of Science and Technology for Development, Cancun, Mexico.

TIMARÁN, R., MILLÁN, M.(2005a), EquipAsso: An Algorithm based on New Relational Algebraic Operators for Association Rules Discovery, in proceedings of the IASTED International Conference on Computational Intelligence (CI 2005), International Association of Science and Technology for Development, Calgary, Canada.

TIMARÁN, R., MILLÁN, M. (2005b), Extensión del Lenguaje SQL con Nuevas Primitivas para el Descubrimiento de Reglas de Asociación en una Arquitecturas Fuertemente Acoplada con un SGBD, en revista Ingeniería y Competitividad, ISSN 0123-3033, Universidad del Valle, Volumen 7, No. 2, Cali, Colombia.

TIMARÁN, R., MILLÁN, M.(2006), New Algebraic Operators and SQL Primitives for Mining Classification Rules, in proceedings of the IASTED International Conference on Computational Intelligence (CI 2006), International Association of Science and Technology for Development, San Francisco, USA.

TIMARÁN, R., (2007), Extensión del Lenguaje SQL con Nuevas Primitivas para el Descubrimiento de Reglas de Clasificación, VI Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento, ISBN 978-9972-2885-1-7, Facultad de Ciencias e Ingeniería, Pontificia Universidad Católica del Perú, Primera Edición, enero de 2007, Lima, Perú.

THOMAS, S., SARAWGI, S. (1998), Mining Generalized Association Rules and Sequential Patterns Using SQL Queries, in Fourth International Conference on Knowledge Discovery and Data Mining, KDD.

YOSHIZAWA, T., PRAMUDIONO, I., KITSUREGAWA, M. (2000), SQL Based Association Rule Mining using Commercial RDBMS (IBM DB2 UDB EEE), Data Warehousing and Knowledge Discovery, pag. 301-306.

WANG, M., IYER, B., SCOTT, V.,J. (1998), Scalable Mining for Classification Rules in Relational Databases, International Database Engineering and Application Symposium, pages 58-67.