Implementación de Primitivas SQL para Reglas de Clasificación en una Arquitectura Fuertemente Acoplada con un SGBD

Ricardo Timarán Pereira, Ph.D.*
ritimar@udenar.edu.co

and

Mario Fernando Guerrero*
mfguerrero@hotmail.com

*Universidad de Nariño, Departamento de Ingeniería de Sistemas
Ciudad Universitaria Torobajo
San Juan de Pasto, Colombia

Abstract

Most KDD systems are loosely coupled with a DBMS. The advantage of this architecture is its portability. Their main disadvantages are poor scalability and performance. A KDD system tightly coupled with DBMS solves these problems, because all the data mining algorithms are integrated into the DBMS engine as a primitive and they are executed jointly with the data.

In this paper, the implementation process of algebraic operators and SQL primitives for the data mining classification task inside the PostgreSQL DBMS, applying the tree-step approach, is presented. This approach facilitates the tightly coupled of a data mining task into the DBMS engine.

Keywords: Knowledge Discovery in Databases, Classification Task, Relational Algebraic Operators, SQL Primitives, Tightly Coupled Architecture.

Resumen

La mayor parte de los sistemas DCBD son débilmente acoplados 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 estos problemas, debido a que todos los algoritmos de minería de datos se integran al motor del SGBD como una primitiva y son ejecutados conjuntamente con los datos.

En este artículo, se presenta el proceso de implementación de operadores algebraicos y primitivas SQL para la tarea de minería de datos clasificación en el interior del SGBD PostgreSQL aplicando el método tres-pasos. Este método facilita el acoplamiento fuerte de una tarea de minería de datos en el motor de un SGBD.

Palabras claves: Descubrimiento de Conocimiento en Bases de Datos, Tarea de Clasificación, Operadores algebraicos, primitivas SQL, Arquitectura Fuertemente Acoplada.

1. Introducción

Muchos investigadores [3] [2] [23] [4] [1] [7] [9] han reconocido la necesidad de integrar los sistemas de descubrimiento de conocimiento y bases de datos, haciendo de esta una área activa de investigación. Los enfoques de integración de sistemas de Descubrimiento de Conocimiento en Bases de Datos (DCBD) y Sistemas Gestores de Bases de Datos (SGBD), reportados en la literatura, se pueden ubicar en uno de tres tipos de arquitectura: sistemas débilmente acoplados, medianamente acoplados y sistemas fuertemente acoplados [24].

La gran mayoría de Sistemas de DCBD son débilmente acoplados con un SGBD. 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 [24]. 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. La ventaja de esta arquitectura es su portabilidad. Sus principales desventajas son la escalabilidad y el 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. Un Sistema DCBD fuertemente acoplado con un SGBD resuelve los problemas de escalabilidad y rendimiento de las otras arquitecturas. En una arquitectura fuertemente acoplada, 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 [24]. Además, todos los algoritmos de minería de datos son ejecutados en el mismo espacio de direccionamiento que los datos en el 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 [30] [33] [31] [32], 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 [7], MSQL [10], Mine Rule [13], OLE DB for Data Mining [15][16], SQL/MM [19] [12], y definiendo nuevas primitivas SQL genéricas que facilitan el proceso de DCBD sin soportar una tarea en particular: NonStop SQL/Mx [5], Count by Group, Count by Order Group [6], FilterPartition, ComputeNodeStatics, PredictionJoin [21].

Otro enfoque de acoplamiento fuerte es el método tres-pasos propuesto en [25][26] y aplicado en [27] para integrar una tarea de minería de datos al interior de un SGBD. El primer paso del método consiste en extender el álgebra relacional con nuevos operadores algebraicos que faciliten los procesos computacionalmente más costosos de la tarea de minería de datos. En el segundo paso, se extiende el lenguaje SQL con nuevas primitivas que se expresan en la cláusula SQL SELECT y que implementan los nuevos operadores algebraicos y en el tercero, se unifican estas primitivas en un nuevo operador SQL, en una nueva cláusula, que extrae el conocimiento deseado por la tarea de minería de datos.

En este artículo se aplica el método tres-pasos para integrar la tarea de clasificación al interior del SGBD PostgreSQL [20][14]. El resultado es un sistema DCBD fuertemente acoplado con el SGBD PostgreSQL para soportar el descubrimiento de reglas de Clasificación en bases de datos.

El resto del artículo se organiza de la siguiente manera. En la sección 2, se presenta los conceptos preliminares acerca de la tarea de clasificación y el modelo de árboles de decisión. En la sección 3, se describe la utilización del *método tres-pasos* para acoplar fuertemente la tarea de clasificación en el SGBD PostgreSQL. En la sección 4, se muestra la manera como se implementaron los operadores algebraicos y las primitivas SQL para clasificación al interior del motor de PostgreSQL. En la sección 5 se reporta el resultado de las pruebas realizadas con las primitivas SQL implementadas y finalmente, en la última sección se presentan las conclusiones y trabajos futuros.

2. Conceptos preliminares

2.1 Tarea de Clasificación

La clasificación de datos es el proceso por medio del cual se encuentran 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. 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 [8]. 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 [8].

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.

2.2 Clasificación por árboles de decisión

El modelo de clasificación basado en árboles de decisión, es, probablemente, uno de los más utilizados por su simplicidad y facilidad para entenderlo [8][21]. Entre los algoritmos de clasificación para árboles de decisión se cuentan ID-3 [17], C4.5 [18], SPRINT [22] y SLIQ [11]. La idea básica de estos algoritmos es la de construir los árboles de decisión en los que:

- Cada nodo interno o no terminal está etiquetado con un atributo.
- Cada rama que sale de un nodo está etiquetada con un valor de ese atributo.
- Cada nodo terminal u hoja representa una clase.
- El nodo inicial o superior se denomina raíz.

Se trata de construir el árbol de decisión más simple que sea consistente con el conjunto de entrenamiento. Para ello, hay que ordenar los atributos relevantes, desde la raíz hasta los nodos terminales, de mayor a menor poder de clasificación. El poder de clasificación de un atributo determinado es su capacidad para generar particiones del conjunto de entrenamiento que se ajuste en un grado dado a las distintas clases posibles, introduciendo de esta forma un orden en dicho conjunto. El orden o el desorden (ruido) de un conjunto de datos es medible. Se han propuesto varias métricas para este proceso. Los algoritmos ID3 [17] y C4.5 [18] utilizan como métrica, para seleccionar el atributo candidato en cada nodo del árbol, la *Ganancia de Información*, mientras que SLIQ [11] y SPRINT [22], usan el indice *Gini*. 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.

3. Método Tres-Pasos para la Integración Fuerte de la Tarea de Clasificación

Aplicando el enfoque de acoplamiento fuerte denominado *método tres-pasos* [25] [26], se han implementado al interior del SGBD PostgreSQL los siguientes operadores algebraicos y las primitivas SQL para soportar la tarea de Clasificación:

3.1 Operadores Algebraicos para Clasificación

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 [33]. Un nuevo operador algebraico para clasificación basado en árboles de decisión debe facilitar el cálculo de las diferentes combinaciones de los atributos condición con el atributo clase y ofrecer operadores agregados específicos, que permitan el cálculo de estas métricas.

En [28][29] se propone el operador algebraico *Mate* que facilita el cálculo de estas combinaciones y los operadores agregados *Entro* y *Gain()* que facilitan el cálculo de las métricas Entropía y Ganancia de Información. Para la generación de las reglas de clasificación se propone el operador *Describe Classifier*.

- *Operador Mate (μ).* Este operador genera por cada una de las tuplas de una relación, todas los posibles combinaciones de los valores no nulos de los *Atributos Condición*, con el valor no nulo del *Atributo Clase*, facilitando el conteo y el posterior cálculo de las medidas de entropía y ganancia de información. El operador *Mate* genera estas combinaciones, en una sola pasada sobre la tabla de entrenamiento.

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

- *Operador agregado Gain*. Este operador permite calcular la reducción de la entropía causada por el conocimiento del valor de un atributo de una relación 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* y $Entro(A_k;Ac;\ R)$ es la entropía de la relación *R* con respecto al atributo condición A_k .

- *Operador Describe Classifier ($\beta\mu$).* Es un operador unario, que toma como entrada la relación resultante de los operadores *Mate*, *Entro* y *Gain* y produce una nueva relación con los valores de los atributos que formarán los diferentes nodos del árbol de decisión. *Describe Classifier* facilita la construcción del árbol de decisión y por consiguiente la generación de reglas de clasificación.

3.2 Primitivas SQL para Clasificación

Los operadores algebraicos *Mate, Describe Classifier* y los operadores agregados *Entro* y *Gain,,* extienden el álgebra relacional para soportar la tarea de clasificación. Estos operadores se implementan en el lenguaje SQL como primitivas dentro de la cláusula SQL SELECT. De esta forma, el lenguaje SQL será capaz de soportar la tarea de clasificación.

- *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 sintaxis:

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

- *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()*.

  La sintaxis de SQL *Entro()* en la cláusula SELECT es la siguiente:

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

- *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()*.

  La sintaxis de *Gain ()* en la cláusula SELECT es:

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

- *Operador Unificado SQL Describe Classification Rules.* Este operador implementa el operador algebraico *Describe Classifier*. *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 cláusula SQL. *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>

Ejemplo1. Sea la tabla CLIENTES(EDAD, INGRESOS, ES_ESTUDIANTE, MANEJOCREDITO,COMPRAEQUIPO) cuyos valores han sido discretizados. Construir el árbol de decisión que permita clasificar a los clientes de acuerdo con los atributos condición *edad, ingresos, es-estudiante, manejocredito* en la clase *compraequipo* con los valores *si* o *no*. Almacenar el resultado en la tabla ReglasClasificación.

La sentencia SQL 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, es_estudiante, manejocredito WITH compraequipo
        GROUP BY edad, ingresos, es_estudiante, manejocredito, compraequipo
```

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.*

4 Aspectos de Implementación de los Operadores Algebraicos y Primitivas SQL para Clasificación en el SGBD PostgreSQL

Con el fin de acoplar fuertemente la primitiva SQL *Mate by* al interior del motor PostgreSQL, se modificaron las estructuras, funciones y nodos en la capa intermedia de la arquitectura de este SGBD. Se modificó el *parser* para que construya, transforme y añada, a las estructuras del compilador, una lista de *atributos condición* y el *atributo clase*. En el *Planner/Optimizer* se agregó un nuevo nodo al *Query tree* denominado *Mate*. Se modificó el *Executor* para que sea capaz de 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 clase. Las funciones agregadas *Entro()* y *Gain()* y el operador *Describe Classification Rules* se implementaron como funciones definidas por el usuario.

4.1 Parser

La implementación del operador *Mate* no modifica el analizador léxico, al no haber nuevos símbolos. Para el análisis sintáctico se realizaron cambios y nuevo código en las funciones o estructuras de los siguientes archivos:

Con el fin de introducir la nueva palabra reservada *Mate By*, se modificó 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 adicionó 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 se muestra en la figura 1. Ahora, en la regla *simple_select* se estableció el punto para activar las nuevas reglas de producción (ver figura 2).

Para recibir y almacenar los atributos condición y el atributo clase del operador *Mate*, se agregó una lista a la estructura *SelectStmnt* en el archivo *…/src/include/nodes/parsenodes.h*.

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

Figura 1. Nuevas reglas de Producción para *Mate By*.

```
+       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 2. Regla *simple_select* con la cláusula *Mate*.

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*. 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 función *transformSelectStmt* del archivo *…/src/backend/parser/analize.c* es la encargada de realizar iniciar la transformación de las estructuras, por tal razón fue necesario incluir una referencia a la función *transformMateClause*. La figura 3 presenta el nodo Query modificado.

4.2 Planner/Optimizar

El Planner, tiene por objetivo la creación de un nuevo plan de ejecución. Para adicionar el operador *Mate* al plan de ejecución fue necesario realizar las siguientes modificaciones:

Se adicionó en la sección Plan Nodes del archivo *…/src/include/nodes/nodes.h* la etiqueta *T_Mate* para reconocimiento del nuevo nodo. En el archivo *…/src/include/nodes/plannodes.h* se incluyó la estructura de datos para este operador. Este nuevo nodo incluye en su estructura un nodo de estado *MateState*, cuya etiqueta *T_MateState* se adicionó en el archivo *…/src/include/nodes/nodes.h* y se definió en *…/src/include/nodes/execnodes.h*. *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.

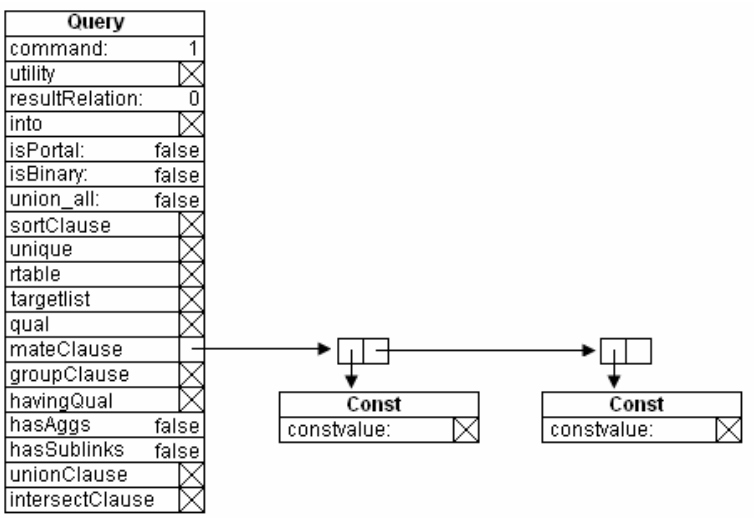

Figura 3. Nodo *Query* con la cláusula *Mate*.

Para crear el nuevo nodo *Mate* con los datos almacenados en el *Query* y los valores actuales del costo de ejecución, se definió la función *make_mate* en el archivo *…/src/include/optimizer/planmain.h* y se implementó en el archivo *…/src/backend/optimizer/plan/createplan.c*. 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 fue necesario incluir una referencia a la función *make_mate*. 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 4 presenta el plan de ejecución de la primitiva *Mate By*.

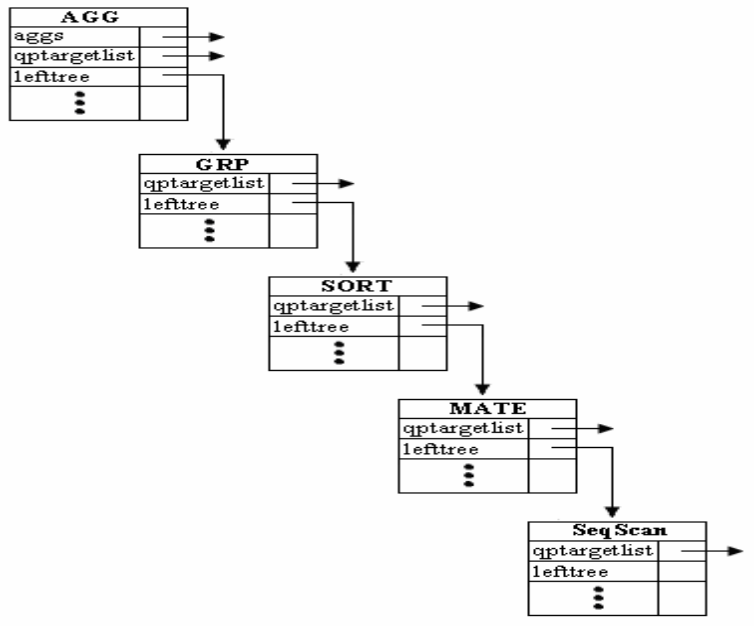

Figura 4. Plan de ejecución de *Mate By*

4.3 Executor

El *executor* es el encargado de ejecutar la estructura *query plan* y de recuperar las tuplas de la forma como lo indicada el plan. Para la ejecución del operador Mate se definieron tres funciones que corresponden a las fases del executor: inicialización (*ExecInitMate*), ejecutación (*ExecMat*e) y finalización (*ExecEndMate*). Se realizaron las siguientes modificaciones:

Se crearon los archivos *…/src/include/executor/nodeMate.h* y *…/src/backend/executor/nodeMate.c* en los cuales se definieron e implementaron las funciones de la interfaz del nodo *Mate* para su ejecución. El archivo *nodeMate.c* contiene las tres funciones básicas del *executor*: *ExecInitMate*, *ExecMat*e y *ExecEndMate*. Estas funciones se muestran en la figura 5.

La función *ExecInitNode* debe configurar el estado de ejecución, la dirección de búsqueda, conjunto de relaciones que intervienen y bloques de memoria disponibles. 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*

La función *ExecMat*e realiza el procesamiento real del plan generado en la etapa anterior y retorna las tuplas que satisfacen el plan. 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.*

La función *ExecEndMate* se encarga de liberar todos los recursos que se reservaron en la ejecución del plan. 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*.

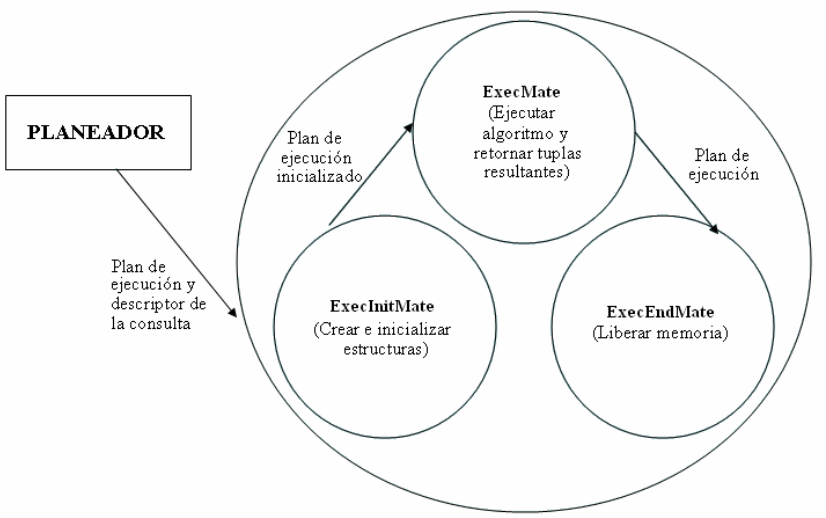

Figura 5. Funciones de manipulación del nodo *Mate*

4.4 Implementación de *Entro*(), *Gain*() y *Describe Classification Rules*()

PostgreSQL tiene la capacidad de crear funciones definidas por el usuario (FDU), las cuales se escriben en el lenguaje de programación anfitrión C o en su lenguaje procedural basado en SQL *PL/pgSQL*. Las funciones agregadas *Entro()* y *Gain()* y el operador *Describe Classification Rules* se implementaron como FDUs utilizando este último lenguaje. Estas funciones se ubican en el directorio: *…/contrib/kdd/clasificacion* como *entro.sql*, *gain.sql* y *Describe_classification_rule.sql.*

5 Pruebas y Evaluación

Para realizar las pruebas con la primitiva *Mate By* y las FDUs *Entro*, *Gain* y *Describe_Classification_Rules* 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. Se utilizaron dos conjuntos de datos: el repositorio de la *UCI (http://www.ics.uci.edu/~mlearn/databases/*), *zoo.data* (5 Kb) con 101 transacciones y 18 atributos con información correspondiente a 101 animales y el conjunto de datos real *udenar.dat* (1.366 Kb) con 20.328 registros, con información correspondiente a la parte académica y social de 20.328 estudiantes de la Universidad de Nariño (Colombia), con los cuales se diseñaron dos pruebas diferentes.

La primera prueba se realizó con el fin de medir el costo computacional de la primitiva *Mate by* y las FDUs *Entro(), Gain() y Describe_ Classification_ Rules().* Con el conjunto de datos *udenar.dat.*, se dejó constante el número de registros y se incrementó el número de atributos, empezando con cinco atributos condición y el atributo clase '*clasepromedio*'. El resultado de esta prueba se muestra en la tabla 1 y la figura 6. El tiempo de las pruebas esta dado en segundos.

No Atributos	Mate By	Entro	Gain	Describe
5	44,507	63,45	57,01	1,47
9	1467,526	6524,01	421,81	2,23
13	3276,637	No aplica	No aplica	No aplica
17				
23				

Tabla 1. Resultados pruebas con la base de datos udenar.dat

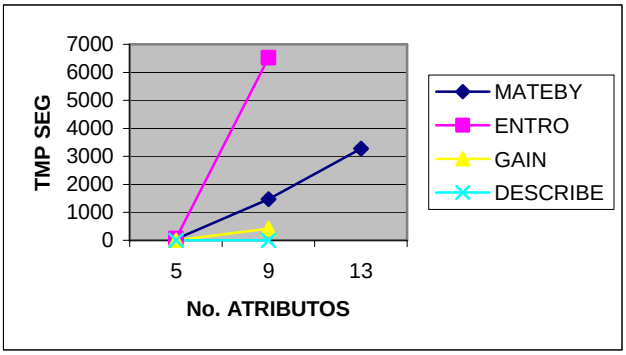

Figura 6. Comportamiento primitiva *Mate by* y FDUs *Entro(), Gain()* y *Describe_classification_rules*().

De acuerdo a la figura 6 la primitiva *Mate by* tiene un comportamiento exponencial y se logra tiempos aceptables hasta los 13 atributos, luego el tiempo de ejecución fue demasiado largo, por lo que no se considero. La función *Entro()* es la mas costosa y solo logró tiempos aceptables hasta los 9 atributos, luego su tiempo es demasiado grande para considerarlo y como de ella dependen *Gain() y Describe_Classification_Rules*(), a partir de los 9 atributos, sus tiempos tampoco se consideran.

La segunda prueba se realizó con el fin de determinar el comportamiento de la primitiva *Mate By* en diferentes tipos de arquitectura y determinar la eficiencia del acoplamiento fuerte. Para ello, se implemento la primitiva *Mate By* en PostgreSQL, como una FDU en el lenguaje PL/pgSQL, en lo que se denomina una arquitectura medianamente acoplada [24]. Se tomó el conjunto de datos *Zoo.data*, duplicando sus registros hasta obtener un máximo de 40.400 transacciones. Se dejó constante el número de registros y se varió el número de atributos iniciando desde 5 y se registraron los tiempos de la primitiva *Mate by* fuertemente acoplada y medianamente acoplada. El tiempo esta dado en milisegundos. Los resultados de esta prueba se muestran en la tabla 2 y en la figura 7.

Como se puede observar en la figura 7, la primitiva *Mate By* se comporta mejor cuando esta acoplada fuertemente con el SGBD PostgreSQL, lo que determina la eficiencia de esta arquitectura con respecto a otras.

6 Conclusiones y Trabajos Futuros

Con este trabajo sobre clasificación y unos anteriores sobre Asociación [25][26][27] se complementa el proyecto PostgresKDD cuyo objetivo es construir una herramienta para el Descubrimiento de Conocimiento fuertemente acoplada con el SGBD PostgreSQL. Estos resultados convierten a PostgreSQL en uno de los primeros SGBD con la

capacidad de compilar y ejecutar una consulta que involucre descubrimiento de conocimiento en grandes volúmenes de datos y capaz de soportar consultas de minería de datos "ad-hoc", i.e, consultas flexibles e interactivas ejecutadas sobre un conjunto de datos con el fin de obtener reglas de asociación y clasificación que cumplan diferentes umbrales.

No. Atributos	***Mate by* Medianamente acoplada**	***Mate by* Fuertemente acoplada**
5	564,77	106,76
6	714,77	226,21
7	1814,77	606,71
8	3294,15	1450,71
9	7192,80	3126,82
10	16055,13	7688,49
11	40499,68	18035,69
12	122241,45	41042,96

Tabla 2. Resultados rendimiento *Mate by* fuerte y medianamente acoplada

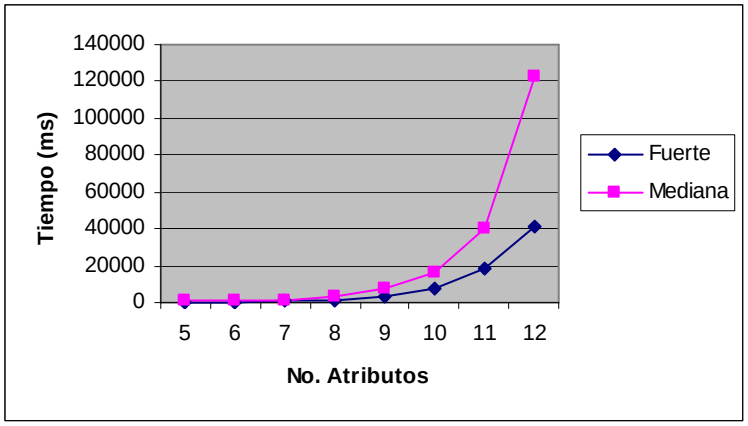

Figura 7. Comportamiento primitiva Mate by fuerte y medianamente acoplada con PostgrSQL.

A partir de las pruebas realizadas se deduce que el tiempo de ejecución de la primitiva *Mate By* crece linealmente con respecto al tamaño del conjunto de datos y exponencialmente con respecto al número de atributos. Por esta razón, y para controlar la explosión combinatoria, y obtener tiempos razonables, el dominio de esta primitiva debe limitarse, por el momento, a problemas con tablas de máximo 20 atributos. Por otra parte, los resultados mostraron que el cálculo de la *entropía* es el proceso más costoso de la tarea de clasificación y que una arquitectura fuertemente acoplada con un SGBD es más eficiente que una medianamente acoplada.

En el futuro, se implementarán en PostgresKDD el resto de operadores algebraicos y primitivas SQL, propuestas en [26], para Asociación y Transformación, con el fin de mejorar este sistema. Se acoplarán fuertemente los operadores algebraicos *Entro* y *Gain* y *Describe Classifier*, actualmente implementados como FDUs, con el fin de mejorar el rendimiento de esta herramienta. Se realizarán mayores pruebas con repositorios reales, tanto en Asociación como en Clasificación con el fin de estudiar la manera de mejor el rendimiento de las primitivas implementadas. Se desarrollarán nuevos operadores algebraicos y primitivas SQL para otras tareas de minería de datos e implementarán en PosgresKDD, con el fin de que soporte todo el proceso de Descubrimiento de Conocimiento en bases de datos. Se implementar otros algoritmos de minería de datos, propuestos por diferentes autores, con el fin comparar rendimientos con los implementados y de ampliar la cobertura de esta herramienta y finalmente, se desarrollará una interfaz gráfica para PostgresKDD que facilite a los usuarios, el proceso de descubrimiento de conocimiento.

Referencias

[1] Agrawal R., Mehta M., Shafer J., Srikant R., Arning A., Bollinger T., The Quest Data Mining System, 2º Conference KDD y Data Mining, Portland, Oregon, 1996.

[2] Botta M., Boulicaut J-F, Masson C., Meo R., A Comparison between Query Languages for the Extraction of Association Rules, Y.Kambayashi, W. Winiwarter, M.Arikawa (Eds): DaWak 2002, LNCS 2454,PP. 1-10,2002.

[3] Boulicaut J-F., Masson C., Data Mining Query Languages, Data Mining and Knowledge Discovery Handbook : A Complete Guide for Practitioners and Researchers, O. Maimon and L. Rokach (Eds), Kluwer Academic Publishers, 14 pages, 2005.

[4] Chaudhuri S., Data Mining and Database Systems: Where is the Intersection?, Bulletin of the Technical Committee on Data Engineering, Vol. 21 No. 1, Marzo, 1998.

[5] Clear, J., Dunn, D., Harvey, B., Heytens, M., Lohman, P., Mehta, A., Melton, M., Rohrberg, L., Savasere, A., Wehrmeister, R., Xu, M., NonStop SQL/MX Primitives for Knowledge Discovery, KDD-99, San Diego, USA, 1999.

[6] Freitas, A.A., Lavington, S.H., 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.

[7] Han, J., Fu Y., Wang W., Koperski K., Zaiane O., 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, 1996.

[8] Han, J., Kamber, M., Data Mining:Concepts and Techniques, Morgan Kaufmann Publishers, San Francisco, California, USA, 2001.

[9] Imielinski, T., Mannila, H., A Database Perspective on Knowledge Discovery, Communications of the ACM, Vol 39, No. 11, November, 1996.

[10] Imielinski T., Virmani A., MSQL: A Query Language for Database Mining, in journal Data Mining and Knowledge Discovery, Kluwer Academica Publishers, Volume 3, Number 4, 1999.

[11] Metha M., Agrawal R., Rissanen J., SLIQ: A Fast Scalable Classifier for Data Mining, Proceedings EDBT96, Avignon, France, 1996.

[12] Melton, J., Eisenberg, A.,SQL Multimedia and Application Packages (SQL/MM), in http://www.sigmod.org/records/issues/0112/standars.pdf, 2001.

[13] Meo R., Psaila G., Ceri S., An Extension to SQL for Mining Association Rules, Data Mining and Knowledge Discovery, Kluwer Academic Publishers, Vol 2, pp .195-224, Boston, 1998.

[14] Momjian, B., PostgreSQL: Introduction and Concepts, Addison-Wesley, May 4, 2000.

[15] Netz A., Chaudhuri S., Bernhardt J., Fayyad U., Integration of Data Mining and Relational Databases, Proceedings of the 26thInternational Conference on Very Large Databases, Cairo, Egytpr,2000.

[16] Netz A., Chaudhuri S., Fayyad U.,Bernhardt J., Integrating Data Mining with SQL Databases: OLE DB for Data Mining, technical report,2000.

[17] Quinlan J.R., Induction of decision trees. Machine Learning, 81-106, 1986.

[18] Quinlan J.R., C4.5: Programs for Machine Leraning, Morgan Kaufmann Publishers, 1993.

[19] Schwenkreis, F., 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), may, 2000.

[20] Stonebraker, M., Rowe, L., A., The Design of POSTGRES, Proceedings of the ACM-SIGMOD Conference, Washington D.C., 1986.

[21] Sattler K, Dunemann O., SQL Database Primitives for Decision Tree Classifiers,CIKM, Atlanta, Georgia, USA, November, 2001.

[22] Shafer J., Agrawal R., Metha M., SPRINT: A Scalable Parallel Classifier for Data Mining, Proceedings of the VLDB Conference, Bombay, India,1996.

[23] Sarawagi S., Thomas S., Agrawal R., Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications, Data Mining and Knowledge Discovery, Kluwer Academic Publishers, Vol 4, 2000.

[24] Timarán, R., 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, Universidad del Valle, Volumen 3, No. 2, Cali, diciembre de 2001.

[25] Timarán R., Millán M., Machuca F., New AlgebraicOperators and SQL Primitives for Mining Association Rules, Proceedings of the IASTED International Conference Neural Networks and Computational Intelligence, Cancun, Mexico, 2003.

[26] Timarán, R., Millán, M., 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, 2005.

[27] Timarán, R., Guerrero, M., Diaz M., Cerquera, C., Armero, S., Implementación de Primitivas SQL para reglas de Asociación en una Arquitectura Fuertemente Acoplada, en proceedings de la XXXI Conferencia Latinoamericana de Informatica (CLEI 2005), Santiago de Cali, Colombia, octubre 2005.

[28] Timarán, R., Millán, M., 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, 2006.

[29] Timarán, R., 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, Lima, Perú, enero de 2007.

[30] Thomas, S., Sarawgi, S., Mining Generalized Association Rules and Sequential Patterns Using SQL Queries, in Fourth International Conference on Knowledge Discovery and Data Mining, KDD, 1998.

[31] Thomas, S., Chakravarthy, S., Performance Evaluation and Optimization of Join Queries for Association Rule Mining, in Proc. Of First Iternational Conference on Data Warehousing and Knowledge Discovery , DAWAK, 1999.

[32] Yoshizawa, T., Pramudiono, I., Kitsuregawa, M, SQL Based Association Rule Mining using Commercial RDBMS (IBM DB2 UDB EEE), Data Warehousing and Knowledge Discovery, pag. 301-306, 2000.

[33] Wang, M., Iyer, B., Scott, V.,J., Scalable Mining for Classification Rules in Relational Databases, International Database Engineering and Application Symposium, pages 58-67, 1998.