# IMPLANTACIÓN DE PRIMITIVAS SQL PARA EL DESCUBRIMIENTO DE REGLAS DE ASOCIACIÓN Y CLASIFICACIÓN AL INTERIOR DEL MOTOR DEL SISTEMA GESTOR DE BASES DE DATOS POSTGRESQL

**STIVENSON ARMERO KREISBERGER**
**CAMILO ANDRES CERQUERA ERAZO**
**MARIO FERNANDO DIAZ**
**MARIO FERNANDO GUERRERO DIAZ**

**UNIVERSIDAD DE NARIÑO**
**FACULTAD DE INGENIERIA**
**PROGRAMA DE INGENIERIA DE SISTEMAS**
**SAN JUAN DE PASTO**
**2005**

# IMPLANTACIÓN DE PRIMITIVAS SQL PARA EL DESCUBRIMIENTO DE REGLAS DE ASOCIACIÓN Y CLASIFICACIÓN AL INTERIOR DEL MOTOR DEL SISTEMA GESTOR DE BASES DE DATOS POSTGRESQL

**STIVENSON ARMERO KREISBERGER**
**CAMILO ANDRES CERQUERA ERAZO**
**MARIO FERNANDO DIAZ**
**MARIO FERNANDO GUERRERO DIAZ**

**Trabajo de Grado presentado como requisito para optar al título de Ingenieros de Sistemas**

**RICARDO TIMARAN PEREIRA**
**Ph.D(C)**
**Director del Proyecto**

**UNIVERSIDAD DE NARIÑO**
**FACULTAD DE INGENIERIA**
**PROGRAMA DE INGENIERIA DE SISTEMAS**
**SAN JUAN DE PASTO**
**2005**

**“Las ideas y conclusiones apartados en esta Tesis de Grado son responsabilidad exclusiva del Autor”**

**Articulo Primero del Acuerdo Número 324 del 11 de Octubre de 1966 emanado del Honorable Consejo Directivo de la Universidad de Nariño**

**Nota de aceptación**

_____
_____
_____
_____
_____
_____

_____
**Firma del presidente del jurado**

_____
**Firma del jurado**

_____
**Firma del jurado**

_____
**Firma del gerente del proyecto**

**San Juan de Pasto, 22 de Junio de 2005**

**A nuestros padres,**
**nuestras familias y amigos,**
**a nuestro director,**
**y a todos aquellos que estuvieron junto a nosotros**
**brindándonos su apoyo y colaboración**
**para culminar este largo camino**
**llenos de éxitos.**

**CONTENIDO**

## LISTA DE TABLAS

## LISTA DE FIGURAS

## LISTA DE ANEXOS

## RESUMEN

Las investigaciones en Descubrimiento de Conocimiento en Bases de Datos (DCBD), se centraron inicialmente en definir modelos de descubrimiento de patrones y desarrollar algoritmos para éstos. Investigaciones posteriores se han focalizado en el problema de integrar DCBD con sistemas de bases de datos, produciendo como resultado el desarrollo de sistemas y herramientas de Descubrimiento de Conocimiento cuyas arquitecturas se pueden clasificar en tres categorías: débilmente, medianamente y fuertemente acopladas con un Sistema de Gestión de Bases de Datos (SGBD).

En este artículo se presenta el proceso de implementación de primitivas SQL para descubrir reglas de Asociación al interior del SGBD PostgreSQL y la evaluación de su rendimiento.



**Palabras claves:** Sistema Gestor de Bases de Datos, Descubrimiento de Conocimiento en Bases de Datos, Tarea de Asociación.

## ABSTRACT

Researchs on Knowledge Discovery in Databases (KDD) was initially oriented toward the definition of new pattern discovery models and development of corresponding algorithms. At present, research has focused on issues related to integrating KDD with database systems, to generate systems and tools for KDD whose architectures can be classified in one of three categories: loosely coupled, middlely coupled and tightly coupled with a Database Management System (DBMS).

In this paper, the implementation process of SQL primitives for mining association rules on top of a PostgreSQL DBMS is presented and its performance evaluation.



**Keywords:** Data Base Management Systems, Knowledge Discovery in Databases, Association Task.

# 1. INTRODUCCIÓN

La implantación de primitivas SQL para el descubrimiento de reglas de Asociación y Clasificación al interior del motor del Sistema Gestor de Bases de Datos (SGBD) PostgreSQL, se presentó como propuesta para proyecto de grado.

En éste capítulo se presenta las generalidades del problema que se identificó como objeto de estudio: La introducción al trabajo de grado realizado, la descripción del problema, los objetivos propuestos los cuales reflejan los alcances del proyecto y la organización del presente documento.

## 1.1 INTRODUCCION

El proceso de extraer conocimiento a partir de grandes volúmenes de datos ha sido reconocido por muchos investigadores como un tópico de investigación clave en los sistemas de bases de datos, y por muchas compañías industriales como una importante área y una oportunidad para obtener mayores ganancias [Tima02a].

El Descubrimiento de Conocimiento en Bases de Datos (DCBD) es básicamente un proceso automático en el que se combinan descubrimiento y análisis. 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. DCBD se puede aplicar en diferentes dominios por ejemplo, para determinar perfiles de clientes fraudulentos (evasión de impuestos), para descubrir relaciones implícitas existentes entre síntomas y enfermedades, entre características técnicas y diagnóstico del estado de equipos y máquinas, para determinar perfiles de estudiantes "académicamente exitosos" en términos de sus características socioeconómicas, para determinar patrones de compra de los clientes en sus canastas de mercado, entre otras.

Las investigaciones en DCBD, se centraron inicialmente en definir nuevas operaciones de descubrimiento de patrones y desarrollar algoritmos para estas. Investigaciones posteriores se han focalizado en el problema de integrar DCBD con Sistemas Gestores de Bases de Datos (SGBD), ofreciendo como resultado el desarrollo de herramientas DCBD cuyas arquitecturas se pueden clasificar en una de tres categorías: débilmente acopladas, medianamente acopladas y fuertemente acopladas con el SGBD[Tima01].

En este proyecto se propone implementar en el interior del motor del Sistema Gestor de Bases de Datos PostgreSQL las Primitivas de Descubrimiento de Conocimiento propuestas por Timarán [Tima02b] que harían mas eficiente el proceso de descubrir reglas de Asociación y Clasificación en una arquitectura fuertemente acoplada con un sistema gestor de bases de datos (SGBD).

## 1.2 DEFINICION DEL PROBLEMA

A pesar del acelerado avance y del incremento de la investigación en el área de DCBD, 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 y herramientas de DCBD existentes se han construido bajo el modelo de arquitectura débilmente acoplada con un SGBD [Tima01].

La actual generación de sistemas de bases de datos se han diseñado principalmente para soportar aplicaciones comerciales. El éxito del lenguaje de consultas SQL se debe al reducido número de primitivas suficientes para soportar una vasta mayoría de aplicaciones. Desafortunadamente, estas primitivas no son suficientes para soportar la emergente familia de nuevas aplicaciones que tratan con el descubrimiento de conocimiento. Existen actualmente herramientas y SGBD que permitan descubrir este conocimiento eficientemente como clementine 9.0 - SPSS (Suite para Data Mining), Random Forest - Salford Systems (para clasificación, clustering y preparación de datos), SAS (para clasificación), Gornik - Gornik System (para clustering y segmentación), Oracle Data Mining 10g Release 2 - Oracle, SQL SERVER 2005 - Microsoft (para clasificación y clustering) entre otros, cuyo elevado costo limita su adquisición y uso; mas aun no existen actualmente SGBD de dominio público que permitan descubrir conocimiento eficientemente.

Timarán [Tima02b] [TiMM03] propone extender el motor de un SGBD con nuevos operadores algebraicos y primitivas SQL que harían mas eficiente el proceso de descubrir reglas de Asociación y Clasificación en una arquitectura fuertemente acoplada con un sistema gestor de bases de datos (SGBD). Estas primitivas SQL no han sido implementadas todavía en ningún SGBD.

Por estas razones se propone implementar estos operadores y primitivas SQL para el descubrimiento de conocimiento en Asociación y Clasificación en el SGBD PostgreSQL. Se ha escogido este SGBD porque es un SGBD de dominio público y se cuenta con el código fuente.

## 1.3 OBJETIVOS

**1.3.1 Objetivo General.** Dotar al SGBD PostgreSQL de la capacidad de Descubrir Conocimiento en Bases de Datos al interior de su motor mediante la implementación de primitivas SQL para el descubrimiento de Reglas de Asociación y Clasificación.

### 1.3.2 Objetivos Específicos.

- Estudiar la literatura concerniente a las generalidades del Descubrimiento de Conocimiento en Bases de Datos y los principios fundamentales de las técnicas de asociación y clasificación de Data Mining.

- Estudiar los operadores de Descubrimiento de Conocimiento propuestas por Timarán [Tima02b] [TiMM03].
- Analizar la Arquitectura del SGBD PostgreSQL y su código fuente.
- Implementar al interior del motor del SGBD PostgreSQL las primitivas SQL para que compile, transforme y ejecute eficientemente una consulta que involucre descubrimiento de reglas de Asociación y Clasificación.
- Evaluar el rendimiento de las primitivas de Asociación y Clasificación implementadas, con bases de datos sintéticas que manejen grandes volúmenes de datos.
- Documentar los cambios realizados a la arquitectura del SGBD PostgreSQL.

## 1.4 ORGANIZACIÓN DEL DOCUMENTO

El resto de este documento esta organizado de la siguiente manera:

- En el segundo capítulo se presenta un estudio sobre el Descubrimiento de Conocimiento en Bases de Datos, especificando las fases del proceso de DCBD, técnicas de Data Mining y las diferentes arquitecturas.
- En el tercer capítulo se presenta un estudio detallado sobre la estructura interna del Sistema Gestor de Bases de Datos PostgreSQL, describiendo las estructuras de datos y cada una de las etapas involucradas en la ejecución de una consulta.
- En el cuarto capítulo se presenta las bases teóricas de las primitivas de asociación y clasificación a ser implementadas.
- En el quinto capítulo se presenta de manera detallada la implementación de las primitivas para el descubrimiento de reglas de Asociación y Clasificación.
- En el sexto capítulo se presenta el análisis de resultados obtenidos en las pruebas de ejecución realizadas sobre las nuevas primitivas.
- En el séptimo capítulo se presenta las conclusiones y recomendaciones obtenidas de éste proyecto.

# 2. PROCESO DE DESCUBRIMIENTO DE CONOCIMIENTO EN BASES DE DATOS

En este capítulo se describe el marco teórico de la investigación en todo lo relacionado con el proceso de DCBD. Los conceptos presentados fueron tomados de [Tima01], [Tima02a] y [TiMM03]. En la primera sección se introducen los conceptos básicos, se define formalmente y se describe las fases del proceso KDD, en la siguiente sección se describe las técnicas más conocidas de Data Mining y finalmente, se describe las Arquitecturas de Descubrimiento de Conocimiento en Bases de Datos.

## 2.1 DESCUBRIMIENTO DE CONOCIMIENTO EN BASES DE DATOS

**2.1.1 Introducción a KDD.** La necesidad de conocimiento es una característica inherente al ser humano y un camino para obtenerlo es a través de la acumulación e interpretación de datos.

El avance de la tecnología en la recolección de los datos como los lectores de código de barras en el comercio y los sensores en la ciencia y la industria, entre otros, generan gran cantidad de datos que se almacenan en las bases de datos, creando una gran demanda por nuevas técnicas y poderosas herramientas que puedan, inteligente y automáticamente, ayudar a transformar estos datos en conocimiento. Ejemplos de este crecimiento de datos son fáciles de encontrar: la mayoría de las transacciones de las Instituciones de Salud son almacenadas en computadoras, con bases de datos del orden de los gigabytes.

El crecimiento desmesurado de datos almacenado en las bases de datos excede las habilidades que tienen los seres humanos para analizar y obtener conocimiento a partir de ellos. Para tratar de descubrir conocimiento no explícito en grandes cantidades de datos, expertos en diversas áreas como Estadística, Inteligencia Artificial y Sistemas de Bases de Datos han propuesto un proceso cuyo objetivo es descubrir relaciones no evidentes entre los datos.

Este proceso, al cual se le ha dado entre otros los nombres de Descubrimiento de Conocimiento en Bases de Datos, Data Mining, Extracción del Conocimiento, Descubrimiento de Información, Arqueología de Datos y Procesamiento de Patrones de Datos, ofrece una alternativa para su extracción.

El término descubrimiento de conocimiento en bases de datos ó KDD, se acuñó en 1989 para referirse al proceso completo de encontrar conocimiento en los datos y para enfatizar el alto nivel de aplicación de métodos de Data Mining.

Data Mining ha sido comúnmente usado por estadísticos, analistas de datos y por la comunidad MIS (Sistema de Administración de Información), mientras que KDD ha sido principalmente usada por investigadores de inteligencia artificial y Aprendizaje Automático. En este documento, el término KDD se refiere al proceso global de descubrir conocimiento útil a partir de los datos, mientras que el de Data Mining se refiere a la etapa en la cual se aplican algoritmos para extraer patrones a partir de los datos.

**2.1.2 Definición de KDD.** El descubrimiento de conocimiento en bases de datos (KDD) se define como el proceso no trivial de identificación de patrones válidos, novedosos, potencialmente útiles y comprensibles a partir de los datos.

El área de descubrimiento de conocimiento sobre bases de datos se forma de la intersección de diferentes campos de investigación. Estos se pueden clasificar en dos grandes grupos: aquellos relacionados con las técnicas de descubrimiento de conocimiento (KD, Knowledge Discovery) y los que se vinculan con el manejo de los datos.

Entre las diversas áreas relacionadas con KDD se incluyen Aprendizaje Automático y adquisición de conocimiento para sistemas expertos, los cuales tratan de usar un modelo del conocimiento que imite, en lo posible, el proceso de aprendizaje humano; Análisis Matemático-Estadístico, en los que se construye un modelo sobre los que se extraen reglas, regularidades y patrones; y Visualización de datos, entre otras.

Un sistema KDD utiliza métodos, algoritmos y técnicas de estos diversos campos. La meta unificada es extraer conocimiento de los datos en el contexto de grandes bases de datos.

Dentro de las áreas que se relacionan con el manejo de los datos, la que más se involucra con KDD, es la de sistema de administración de bases de datos, la cual se concentra en mejorar la eficiencia y la escalabilidad en el almacenamiento y manejo de grandes cantidades de datos. Otra área importante es el Data Warehousing, el cual enfrenta los problemas de almacenamiento y recuperación de información útil para sistemas de soporte a la decisión, más que para propósitos operacionales de bajo nivel.

KDD es un proceso que se compone de diferentes pasos, entre los que se incluyen la selección de los datos, la discretización de valores, el refinamiento de los datos, la búsqueda de patrones y la evaluación del conocimiento extraído.

Los patrones que se obtiene del proceso deben ser válidos con algún grado de certeza y novedosos al menos para el usuario que los requiere. Los patrones deberían ser comprensibles para el usuario final, bien directamente o bien después de algún tipo de preprocesamiento.

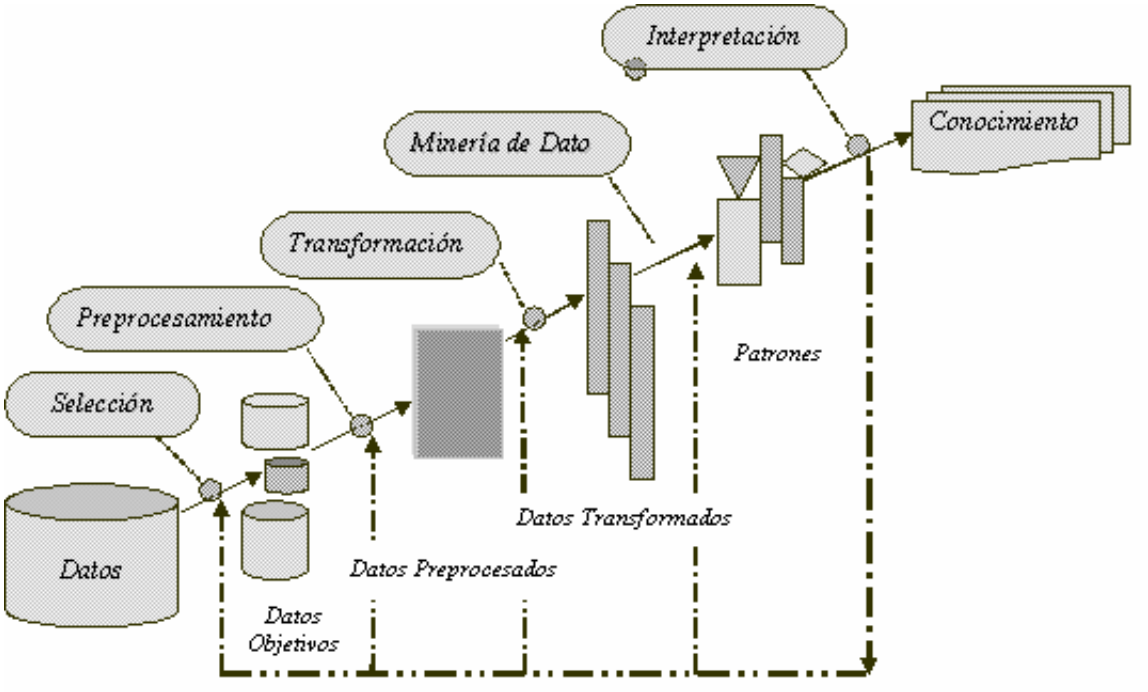
Data Mining es un paso en el proceso de KDD que consiste en la aplicación de algoritmos particulares que, bajo algunas limitaciones de eficiencia computacional, produce un conjunto de patrones.

El componente de Data Mining en el proceso de KDD se interesa principalmente por el significado de los patrones los cuales son extraídos a partir de los datos. El descubrimiento del conocimiento involucra la evolución y posiblemente la interpretación de patrones para tomar la decisión de qué constituye conocimiento.

**2.1.3 Fases del Proceso KDD.** El proceso de KDD es interactivo e iterativo, involucra numerosos pasos con muchas decisiones realizadas por el usuario que debe proporcionar los atributos que mejor modelizan los datos a estudiar.

El proceso de descubrimiento de conocimiento en Bases de Datos consta de las siguientes fases:

**Figura 2.1. Fases del proceso KDD**




1. Definición del problema: Lo primero que se debe hacer es definir cuál es la meta del proceso KDD desde el punto de vista del cliente.

2. Creación de la base de datos de explotación (Selección): Una vez se halla definido el problema, se debe seleccionar los datos a estudiar. Generalmente, los datos que se necesitan no están en una única base de datos, sino que están dispersos por varias fuentes distintas, cada una de ellas con sus propias tablas y sus propios atributos. Teniendo en cuenta la gran cantidad de tablas que puede haber en las distintas bases

de datos, no es difícil darse cuenta de lo costoso en tiempo que puede ser esta elección.

3. Limpieza (Preprocesamiento): Cuando los datos relevantes para el proceso ya han sido almacenados, habrá que estudiarlos. Primero, porque es muy posible que existan datos redundantes, ya que puede haber atributos que representen el mismo concepto aún cuando no tengan el mismo nombre. Segundo, porque al integrar datos de tantas fuentes diferentes es posible que aparezcan valores nulos, registros repetidos o atributos que no sirven en el proceso KDD (por ejemplo, los atributos llaves). Tercero, porque algunos atributos se deben discretizar.

4. Transformación (Reducción): Permite llevar a cabo estrategias para representar los datos dependiendo de los requerimientos del usuario. Se utiliza reducción dimensional tanto vertical (eliminando atributos) como horizontal (eliminando filas duplicadas) para reducir el número de variables y la representación del conocimiento.

5. Minería de Datos: En esta fase se elige el algoritmo a ejecutar para extraer los patrones. Posteriormente estos patrones se deben evaluar e interpretar.

6. Interpretación y Evaluación: Es muy frecuente que el usuario decida explorar los datos para verificar la respuesta obtenida, y por lo tanto, ejecutará el algoritmo nuevamente. De este modo, se entra en una fase de refinado iterativo de los patrones. Si el usuario descubre que necesita nuevos datos, es posible que añada otros parámetros al algoritmo o incluso en algunos casos se añadan nuevos atributos. Tras varias iteraciones, el usuario puede determinar si los resultados son lo suficientemente buenos y pondrá los patrones en uso.

## 2.2 DATA MINING

La definición más generalmente aceptada de Data Mining es “el proceso de búsqueda de patrones, tendencias o relaciones, en definitiva conocimiento, en datos detallados”.

Las técnicas de Data Mining se usan generalmente para describir datos. La descripción pretende encontrar una representación de los datos fácilmente entendible por los seres humanos. Los seres humanos tienen muchas dificultades para poder extraer conclusiones de cantidades de datos tan grandes como las que están siendo tratadas, por lo que se necesita alguna forma de representarlos, ya sea mediante árboles, reglas, o cualquier otra forma de representación del conocimiento.

Las tareas primitivas que se pueden realizar, según el tipo de conocimiento que se requiera extraer, son:

**Predicción:** Es una tarea de aprendizaje de patrones por medio de ejemplos, utilizando un modelo de desarrollo para predecir valores futuros de una variable designada.

**Caracterización:** Es la tarea de abstraer las características comunes de un conjunto de datos para poder describir cada grupo o clase. Por ejemplo, se pueden extraer patrones de llamadas en una empresa de telecomunicaciones o describir los síntomas de una enfermedad. Existen muchos métodos para extraer reglas de este tipo, entre los que se destacan la inducción orientada a atributos y la teoría de los Rough Sets.

**Discriminación:** Consiste en extraer las características comunes de un grupo de datos, a los que se denominarán clase objetivo, que los distinguen con un factor de error de los miembros de otros grupos, que se denominarán clases discriminantes. Por ejemplo, se pueden encontrar los síntomas de una enfermedad que la distinguen totalmente del resto de enfermedades.

**Clasificación:** Consiste en determinar la pertenencia de un objeto a un grupo de entre varios. Se suministra un conjunto de datos pertenecientes a diversas clases. A partir de ellos, se extraen las características que permitirán clasificar los nuevos datos en sus correspondientes clases. Algunos sistemas que permite extraer reglas de este tipo son ID3 o C4.5.

**Asociación:** Es el proceso de descubrir relaciones o asociaciones entre los datos. Generalmente, estas reglas se buscan en las llamadas bases de datos transaccionales. Como ejemplo de aplicación de este tipo de algoritmos, se tiene el caso del análisis de la canasta familiar. En estas bases de datos, los nombres de los atributos son nombres de productos, cuyo valor es 0, si no está presente en la transacción, o 1 si está presente. De este modo, cada fila representa una "compra" determinada del cliente, y se trata de buscar relaciones entre los objetos de la transacción, es decir, siempre que se compra A, se compra B. Por ejemplo siempre que se compra Arroz se compra Aceite.

**Clustering o Segmentación**: Consiste en dividir la población en clases o "clusters" con el criterio de minimizar la distancia entre elementos de una clase y maximizar la distancia entre clases. Una vez que todos los grupos han sido determinados, se extraen las características de cada cluster. Por ejemplo, se pueden agrupar piezas de maquinaria según su forma, y una vez que quedan determinados los clusters se pueden utilizar para describir ese conjunto de piezas.

**Patrones Secuenciales:** Consiste en descubrir relaciones entre los datos, que se recogen durante un periodo de tiempo. De este modo, se puede determinar si dado un suceso A, ocurrirá un suceso B después de n meses. Básicamente se trata de una variante de la consulta de asociación en las cuales el factor tiempo interviene de forma relevante.

**Series Temporales:** Es el proceso descubrir similitudes entre diferentes conjuntos de datos que evolucionan en el tiempo, para tratar de identificar patrones de comportamiento. De este modo, se pueden identificar compañías con patrones de crecimiento similar, o averiguar si una partitura musical es similar a otra en algún fragmento.

**2.2.1 Técnicas de Data Mining.** En esta sección se describe únicamente dos técnicas Data Mining: algoritmos de Reglas de Asociación y el algoritmo de Clasificación basados en el método de árboles de decisión C4.5.

**2.2.1.1 Reglas de Asociación.** Este enfoque de Data Mining consiste en encontrar un conjunto de reglas que identifiquen patrones de información ocurridos frecuentemente en una base de datos.

Agrawal, Imielinski, y Swami introdujeron las reglas asociativas, y proporcionaron el algoritmo AIS para encontrarlas. El dominio de aplicación de estas reglas va desde el soporte a la toma de decisiones hasta el diagnóstico y predicción de alarmas en telecomunicaciones. Otras aplicaciones incluyen el diseño de catálogos, segmentación de mercado, y patrones de compra.

En general, una regla de asociación se define como sigue: Sea $I = \{i1, i2, i3,..., im\}$ un conjunto de atributos sobre el domino binario {0,1}, llamado ítem. Sea D una base de datos transaccional, en la cual cada transacción d está representada como un vector binario, con d[k]=1 si d compra el articulo $d_k$, y d[k]=0 en caso contrario; donde cada transacción d está en el conjunto de ítems, tal que $d \subseteq I$. Una regla de asociación es una implicación de la forma $\mathbf{X \rightarrow Y}$, donde $X \subseteq I$, $Y \subseteq I$, y $X \cap Y = \phi$ , $X \rightarrow Y$ ocurre en el conjunto de transacciones D con confianza c, si el c% de las transacciones en D que contiene a X también contiene a Y. La regla tiene soporte s en D si s% de las transacciones en D contiene $X \cup Y$.

Dado un conjunto de transacciones D, el problema de reglas de asociación es descubrir todas las reglas que tienen un soporte y una confianza mayor que la especificada por el usuario.

En [AS94] se propone el algoritmo Apriori y AprioriTid para resolver el problema, y se combinan dando como resultado un algoritmo Apriori híbrido que reúne las mejores características de los anteriores.

Por simplicidad, los autores asumen que los itemset en una transacción están ordenados lexicográficamente y que el tamaño de un itemset corresponde al número de ítems contenidos en él. Un itemset de tamaño k es entonces un k-itemset. Se usa la notación c[1], c[2], c[3],... c[k] para representar un k-itemset que consiste de los ítems c[1], c[2],... c[k], donde c[1] < c[2]<... < c[k]. Asociado a cada itemset existe un contador que se usa para calcular su soporte, es decir, el número de transacciones diferentes de la base de datos que lo contiene. En la Tabla 2.1 se presenta la notación usada en el algoritmo Apriori y sus variantes:

**Tabla 2.1. Notación empleada en el algoritmo Apriori.**

| k-itemset | Un itemset contiene k ítems |
|---|---|
| $L_k$ | Conjunto de large k-itemsets (que cumplen con el mínimo soporte)<br>Cada elemento de estos conjuntos es una pareja (itemset, contador del soporte). |
| $C_k$ | Conjunto de candidatos k-itemset (itemset potencialmente large).<br>Cada elemento de estos conjuntos es una pareja (itemset, contador del soporte). |

**2.2.1.1.1 Algoritmo Apriori.** En el primer paso del algoritmo Apriori se calcula el número de ocurrencias de cada ítem en todas las transacciones registradas en la base de datos, para determinar los large 1-Itemsets. En el segundo paso, el algoritmo entra en un ciclo donde para cada iteración k se realizan fundamentalmente dos procesos. En el primero, los large Itemsets $L_{k-1}$ encontrados en el paso k-1 son usados para generar los Itemsets candidatos del paso actual ($C_k$), usando la función apriori_gen(), que a su vez almacena los candidatos generados en un árbol hash. En el segundo proceso, la base de datos se recorre para determinar cuales Itemsets aparecen en cada transacción e incrementar así el soporte de los $C_k$ en el árbol hash. Al finalizar el recorrido de la base de datos, la información almacenada en el árbol hash y el valor de minsup dado por el usuario, se usan para determinar cuales Itemsets son large Itemsets y cuales no. El algoritmo Apriori se presenta en la Figura 2.2.

**Figura 2.2. Algoritmo Apriori**

```
C_k
L_1 = {large 1- Itemsets};
For (k=2; L_{k-1} ≠ ∅; k++) do begin
        C_k = apriori_gen(L_{k-1}); //Nuevos Candidatos
        Forall transacciones t ∈ D do begin
                C_t = Subconjunto(C_k, t); //Candidatos contenidos en t
                For all candidatos c ∈ C_t do
                        c.count ++
                End
        End
        L_k = {c ∈ C_k | c.count ≥ minsup}
End
Respuesta = ∪k L_k
```

Haciendo uso de la propiedad de Cláusula hacia abajo (Propiedad que garantiza que cualquier subconjunto de un large Itemset, es un large Itemset), en la generación del conjunto de k-Itemsets candidatos no se tiene en cuenta los k-Itemsets cuyos subconjuntos

de (k-1)-Items no sean todos large Itemsets, reduciéndose así el espacio de búsqueda. Los autores también resaltan la forma óptima como usan el árbol hash para calcular los k-large Itemsets en una transacción t, basados en los $C_k$.

**2.2.1.1.2 Generación de Reglas.** Las reglas se calculan de la siguiente manera: para cada ítem frecuente, se generan todas las reglas $a \rightarrow (l - a)$, donde a es un subconjunto de l tal que $r = \frac{soporte(l)}{soporte(a)}$ sea mayor o igual que la confianza mínima. El soporte de cualquier subconjunto $\bar{a}$ de a tiene que ser mayor que el soporte de a. Además, la confianza de la regla $\bar{a} \rightarrow (l - \bar{a})$ no puede ser mayor que la confianza de $a \rightarrow (l - a)$.

Por lo tanto, si a no produce ninguna regla que involucren todos los ítems en l y además no la contenga, entonces, no habrá tampoco ninguna regla que contenga como antecedente algún subconjunto de a. De esto se sigue que para una regla $(l - a) \rightarrow a$ confiable, todas las reglas de la forma $(l - \bar{a}) \rightarrow \bar{a}$ deben ser confiables, siempre y cuando $\bar{a}$ no sea el subconjunto vació de a. Por ejemplo, si la regla $AB \rightarrow CD$ es confiable, entonces las reglas $ABC \rightarrow D$ y $ABD \rightarrow C$ deben ser confiables. Esta característica de confianza de las reglas es análoga a la propiedad de los ítems frecuentes, donde si un itemset es frecuente lo serán todos los subconjuntos del mismo. De un itemset frecuente l, primero se generan todas las reglas con ítem en el consecuente. Usando las reglas confiables de un solo consecuente se generan entonces las posibles reglas con dos ítems en el consecuente, generadas a partir de l, y así consecutivamente.

**2.2.1.2 Clasificación.** La tarea de clasificación consiste en distribuir el conjunto de datos en un número de categorías posibles definido apriori por un experto.

La tarea de clasificación es supervisada, es decir, es necesario especificar las características y el universo de las distintas clases, además de tener que proporcionarle un conjunto preparado de datos, de manera que el sistema pueda crear una estructura que permita aprender y generalizar para determinar la clase para nuevas características.

Existen varios métodos de clasificación desarrollados en diversas áreas como aprendizaje de máquina, bases de datos, estadística, redes neuronales y Rough Sets. Estas técnicas generan modelos de clasificación en forma de árboles de decisión o en forma de reglas de clasificación.

**2.2.1.2.1 Árboles de Decisión.** El aprendizaje por árboles de decisión es uno de los métodos más sencillos, fáciles de implementar y a su vez de los más poderosos para inferencia inductiva. Son usados con éxito en diversas áreas investigativas, tales como Reconocimiento de Caracteres, Reconocimiento de Voz, Diagnóstico Médico, Sistemas

Expertos y Señales de Radar. Es un método para aproximación de valores discretos, en el cual la función de aprendizaje se representa por medio de un árbol. El aprendizaje también se puede representar como un conjunto de reglas SI – ENTONCES para mejorar su comprensión.

Un problema fundamental en la construcción de los árboles de decisión radica especialmente en hallar árboles que clasifiquen correctamente todo el conjunto objeto (registros de la Base de Datos).

El despliegue gráfico y la facilidad de interpretación son quizás, en parte, responsables de la popularidad de los árboles de decisión, pero sus rasgos más importantes son su naturaleza jerárquica y su flexibilidad.

La naturaleza jerárquica de los árboles de decisión es uno de sus rasgos básicos, su jerarquía por niveles lo convierte en un excelente clasificador de objetos. En cada nivel del árbol se encuentra una selección minuciosa del conjunto de objetos. La flexibilidad en su estructura le permite acoplarse fácilmente a cualquier cambio presentado en el conjunto de objetos de entrada.

**A continuación se presentan los conceptos básicos para abordar el proceso de construcción de un árbol de decisión:**

Un árbol está compuesto por una dupla G = (N, A), donde N es el conjunto de nodos y A es el conjunto de arcos.

Un camino de un árbol, es una sucesión de arcos de la forma $(a_1, a_2)$, $(a_2, a_3)$,..., $(a_{n-1}, a_n)$. Se dice que el camino de $a_1$ hasta $a_n$ es de longitud n.

Los arcos son una tripleta de la forma $(n_i, n_j, e_{ij})$ los cuales establecen una relación entre los nodos $n_i$, $n_j$ de N, en un sentido $(n_i \rightarrow n_j)$, y una etiqueta asociada $(e_{ij})$.

El árbol debe cumplir las siguientes propiedades:

- Hay solamente un nodo raíz, al cual ningún arco llega.
- Cada nodo, excepto el raíz tiene solamente un arco de llegada.
- Hay un único camino del nodo raíz a cada nodo.

Si $(n_k , n_z)$ son nodos en un árbol, entonces $n_k$ es el padre de $n_z$ y $n_z$ es el hijo de $n_k$. Si hay un camino de $n_k$ a $n_z$ $(n_k \neq n_z)$, entonces $n_k$ es el predecesor de $n_z$ y $n_z$ es el sucesor de $n_k$.

Un nodo sin sucesor, es un nodo hoja o terminal, todos los demás nodos, excepto el raíz, son nodos interiores. La profundidad de un nodo $n_i$ en un árbol, es la longitud desde el nodo raíz hasta el nodo $n_i$.

En la tarea de formación del árbol interviene un Conjunto de Objetos (universo) U≠∅, este conjunto está conformado por una dupla S = {T, V}, donde T representa un conjunto de atributos y V todos los posibles valores disjuntos que toman los atributos.

Esta dupla se puede representar como una tabla cuyas columnas y filas están conformadas por atributos y por el conjunto de valores que toma cada atributo respectivamente. Los posibles valores que toman los atributos se pueden clasificar en: nominales, categóricos ó numéricos.

El conocimiento obtenido es expresado en forma de Reglas de Clasificación. Estas reglas se expresan en forma de un par (condición-acción) $C \Rightarrow A$, donde C es el conjunto de atributos de condición y A es el atributo decisión o clase. En general, un árbol representa una disyunción de conjunciones de restricciones en los valores de los atributos. Cada camino desde la raíz del árbol hasta una hoja corresponde a una conjunción de atributos de prueba, donde las hojas del árbol se constituyen en las clases, y el árbol por si mismo es una disyunción de estas conjunciones.

El proceso de aprendizaje del árbol, se considera bueno si permite predecir satisfactoriamente las clasificaciones a las que pertenecen los objetos. En la construcción del modelo de clasificación se utilizan una buena cantidad de objetos. Este conjunto de objetos U es dividido en dos partes: los datos de entrenamiento L1 y los datos de prueba L2 [OJ01].

Datos de entrenamiento: Estos datos se obtienen especialmente del conjunto de objetos y sirven para realizar los ajustes de aprendizaje del algoritmo así como para probar el conocimiento obtenido.

Datos de Prueba: Estos datos permiten establecer si el proceso de aprendizaje del árbol puede clasificar nuevos datos con certeza o no. La metodología utilizada consiste en separar los datos de entrenamiento de los datos de prueba, para lograr de esta forma una mayor predicción y exactitud en la información final suministrada.

**2.2.1.2.2 Algoritmo C4.5.** Un posible enfoque de la tarea de inducción podría ser generar todos los posibles árboles de decisión que clasifican correctamente el conjunto de datos de entrenamiento y elegir el mejor de todos ellos. La cantidad de árboles que se generarían es finito pero grande, con lo que, esta posibilidad solo es aceptable cuando se trata con conjuntos pequeños de datos.

El algoritmo C4.5 fue pensado justo para los casos en los que se tiene gran cantidad de objetos y atributos y se requiere la construcción de un árbol de clasificación sin mucho tiempo computacional. La estructura básica del algoritmo es iterativa. Un subconjunto del conjunto de datos inicial denominado datos de entrenamiento es elegido aleatoriamente, y se utiliza para la construcción del árbol. Como resultado se obtiene un árbol capaz de

clasificar todos los elementos del conjunto de entrenamiento. En este caso el proceso termina, en caso contrario, se añaden los objetos no clasificados nuevamente a los datos de entrenamiento y el proceso continua. El problema por tanto, radica en los criterios para el cálculo del árbol con una cantidad arbitraria de objetos. El punto crucial en el cálculo del árbol está en la elección del mejor atributo. C4.5 utiliza la entropía de Shannon para tomar la decisión, cuyo costo computacional para cada iteración en C4.5, es proporcional al producto del tamaño del conjunto de entrenamiento, el número de atributos y el número de nodos que no son hojas en el árbol. C4.5 consta de un sistema de aprendizaje que obtiene reglas de clasificación y que utiliza árboles de decisión como modelo de representación. El algoritmo se aplica al conjunto de datos de entrenamiento que se puede encontrar como un archivo plano.

El núcleo del algoritmo C4.5 es el proceso de construcción del árbol de decisión. Los pasos básicos del algoritmo para la construcción del árbol son:

- Calcular la ganancia de información de cada atributo.
- Seleccionar el atributo que da mayor la ganancia de información y utilizarlo como el elemento de decisión en el nodo en el que se encuentre del árbol.
- Ramificar el nodo con todos sus posibles valores.
- Reagrupar los datos en las ramas correspondientes.
- Repetir el proceso para cada rama del árbol.

El algoritmo finaliza cuando todos los objetos en cada nodo pertenecen a la misma clase. Los detalles precisos de la construcción del árbol varían según las distintas implementaciones.

El punto crítico de esta estrategia de generación de árbol, es la elección del atributo de partición de cada nodo. El proceso de selección de cada atributo se fundamenta en una función heurística que minimiza la medida de entropía aplicada a los ejemplos en un nodo. Esta función heurística permite seleccionar el atributo que provee la mayor ganancia de información (Information Gain), es decir, el atributo que minimiza la información necesaria en los subárboles resultantes para clasificar los elementos. La medida de entropía favorece los atributos que particionan los datos en subconjuntos que tienen baja entropía de clase. Un subconjunto de datos tiene una baja entropía de clase cuando la mayoría de ejemplos pertenecen a una sola clase, C4.5 elige el atributo que posee el grado máximo local de discriminación entre clases [OJ01].

**2.2.1.2.2.1 Entropía.** La palabra entropía parte de la palabra griega *e tropé* que significa transformación y es muy parecido al término energía.

A mediado de los años 40's Claude Shannon publica su artículo “Teoría de la información de sistemas de comunicación”. La teoría de la información es un tema matemático que trata con tres conceptos básicos:

- La medida de información.
- La capacidad de un canal de comunicación para transferir información.
- La codificación como un medio de utilizar los canales a toda su capacidad.

Shannon consiguió determinar con unidades el concepto de información, creando así la medida de "Cantidad de Información". Esto conduce a expresar que la medida de la información está relacionada con la Incertidumbre.

La teoría de la información no representa una herramienta para manipular la incertidumbre, pero si para medirla. Desde este punto de vista, la cantidad de conocimiento (información) y la pérdida de esta (incertidumbre) se complementan.

La incertidumbre inherente a la solución de un problema proporciona información deficiente, la cual puede ser imprecisa, incompleta o en alguna forma contradictoria. La incertidumbre simboliza la incompletitud e inexactitud del conocimiento sobre las características del ambiente. Si se puede medir la cantidad de incertidumbre inherente a una situación y reducirla hasta obtener información relevante, entonces la cantidad de información que se obtiene mediante este proceso puede cuantificar una medida, la cual disminuye la incertidumbre resultante.

Cuando existen varios espacios de probabilidades, a cada estado se le asocia una probabilidad, es decir, a una variable aleatoria se le asocia su espacio de probabilidades. Se define entonces la *Cantidad de Información* de un estado *i* como:

$$I_i = -\log_2(P_i)$$

Siendo $P_i$ la probabilidad asociada al estado *i.* El signo menos es para hacer que $I_i$ siempre sea positivo, ya que $P_i$ siempre estará entre 0 y 1, y su logaritmo es negativo.

Con la suma de todas las cantidades de información de todos los estados posibles queda:

$$I = \sum_{i=1}^{n} I_i = \sum_{i=1}^{n} p_i * (-\log_2(P_i)) = -\sum_{i=1}^{n} p_i * \log_2(P_i)$$

A la anterior fórmula se le conoce en el mundo de la teoría de la información como Entropía, denotándose comúnmente por la letra *H*:

$$H = -\sum_{i=1}^{n} p_i * \log_2(P_i)$$

La fórmula de Entropía utiliza el ***log***$_2$ (logaritmo base 2) como medio para contrarrestar el carácter exponencial de los estados posibles y porque la entropía es una medida de la codificación en bits.

**2.2.1.2.2.2 Ganancia de Información (Information Gain).** El proceso de selección de los atributos en el algoritmo C4.5, se fundamenta en una heurística (Information Gain) que minimiza la medida de entropía aplicada a los ejemplos en un nodo.

Esta heurística permite seleccionar el atributo que provee la mayor ganancia de información, es decir, el atributo que minimiza la información necesaria en los subárboles resultantes para clasificar los elementos. La medida de entropía favorece los atributos que particionan los datos en subconjuntos que tienen baja entropía de clase. Un subconjunto de datos tiene baja entropía de clase cuando la mayoría de ejemplos pertenecen a una sola clase. C4.5 básicamente elige el atributo que posee el grado máximo local de discriminación entre clases.

El objetivo central del algoritmo C4.5 es seleccionar el atributo a probar en cada nodo del árbol. Se debe seleccionar el atributo que tenga una mayor clasificación sobre el conjunto de objetos de decisión S. C4.5 usa la medida de Ganancia de Información para seleccionar el atributo candidato en cada etapa mientras va creciendo el árbol. La ganancia de información, Gain(S, A) de un atributo A, con respecto a un conjunto de objetos S, se define como:

$$Gain(S, A) = Entropia(S) - \sum_{v \in Valores(A)} \frac{|S_v|}{|S|} Entropia(S_v)$$

Sea la tabla 2.2 un conjunto de datos de entrenamiento, en el cual se han registrado las mañanas de los sábados de acuerdo con su adecuación para Jugar Tenis:

**Tabla 2.2. Conjunto de datos de entrenamiento**

| *Día* | *Pronostico* | *Temperatura* | *Humedad* | *Viento* | *Jugar Tenis* |
|---|---|---|---|---|---|
| *D1* | *Soleado* | *Caliente* | *Alta* | *Débil* | *No* |
| *D2* | *Soleado* | *Caliente* | *Alta* | *Fuerte* | *No* |
| *D3* | *Nublado* | *Caliente* | *Alta* | *Débil* | *Si* |
| *D4* | *Lluvia* | *Templado* | *Alta* | *Débil* | *Si* |
| *D5* | *Lluvia* | *Frío* | *Normal* | *Débil* | *Si* |
| *D6* | *Lluvia* | *Frío* | *Normal* | *Fuerte* | *No* |
| *D7* | *Nublado* | *Frío* | *Normal* | *Fuerte* | *Si* |
| *D8* | *Soleado* | *Templado* | *Alta* | *Débil* | *No* |

| | | | | | |
|---|---|---|---|---|---|
| *D9* | *Soleado* | *Frío* | *Normal* | *Débil* | *Si* |
| *D10* | *Lluvia* | *Templado* | *Normal* | *Débil* | *Si* |
| *D11* | *Soleado* | *Templado* | *Normal* | *Fuerte* | *Si* |
| *D12* | *Nublado* | *Templado* | *Alta* | *Fuerte* | *Si* |
| *D13* | *Nublado* | *Caliente* | *Normal* | *Débil* | *Si* |
| *D14* | *Lluvia* | *Templado* | *Alta* | *Fuerte* | *No* |

Sea S la clase de decisión (jugar-tenis), formado por 14 ejemplos los cuales incluye 9 objetos positivos (*Si*) y 5 objetos negativos (*No*). La entropía de S es:

$$Entropia = ([9+,5-]) = H(S) = -\left(\frac{9}{14}\right)\log_2\left(\frac{9}{14}\right) - \left(\frac{5}{14}\right)\log_2\left(\frac{5}{14}\right) = 0.940$$

A continuación se calcula la *Ganancia de Información* de los atributos condición con respecto a la *Clase Decisión*.

Iniciamos con el atributo *Pronóstico*:

Valores (*Pronóstico*) = Soleado, Nublado, Lluvioso

$$Gain(S, \text{Pronóstico}) = Entropia(S) - \sum_{v \in \{Soleado, Nublado, Lluvioso\}} \frac{|S_v|}{|S|} Entropia(S_v)$$

$$= Entropia(S) - \left(\frac{5}{14}\right) Entropia(S_{Soleado}) - \left(\frac{4}{14}\right) Entropia(S_{Nublado}) - \left(\frac{5}{14}\right) Entropia(S_{Lluvioso})$$

$$= 0.940 - \left(\frac{5}{14}\right) * (0.97) - \left(\frac{4}{14}\right) * (0.00) - \left(\frac{5}{14}\right) * (0.97)$$

$$= 0.246$$

Ahora se calcula la Ganancia para el atributo *Temperatura*:

Valores (*Temperatura*) = Caliente, Templado, Frío

$$Gain(S, Tempratura) = Entropia(S) - \sum_{v \in \{Caliente, Templado, Frio\}} \frac{|S_v|}{|S|} Entropia(S_v)$$

$$= Entropia(S) - \left(\frac{4}{14}\right) Entropia(S_{Caliente}) - \left(\frac{6}{14}\right) Entropia(S_{Templado}) - \left(\frac{4}{14}\right) Entropia(S_{Lluvioso})$$

$$= 0.940 - \left(\frac{4}{14}\right) * (1.00) - \left(\frac{6}{14}\right) * (0.918) - \left(\frac{4}{14}\right) * (0.811)$$

$$= 0.029$$

La Ganancia para el atributo *Humedad* es:

Valores (*Humedad*) = Alto, Normal

$$Gain(S, Humedad) = Entropia(S) - \sum_{v \in \{Alto, Normal\}} \frac{|S_v|}{|S|} Entropia(S_v)$$

$$= Entropia(S) - \left(\frac{7}{14}\right) Entropia(S_{Alto}) - \left(\frac{7}{14}\right) Entropia(S_{Normal})$$

$$= 0.940 - \left(\frac{7}{14}\right) * (0.985) - \left(\frac{7}{14}\right) * (0.592)$$

$$= 0.151$$

Finalmente, la ganancia para el atributo *Viento* es:

Valores (*Viento*) = Débil, Fuerte

$$Gain(S, Viento) = Entropia(S) - \sum_{v \in \{Débil, Fuerte\}} \frac{|S_v|}{|S|} Entropia(S_v)$$

$$= Entropia(S) - \left(\frac{8}{14}\right) Entropia(S_{Debil}) - \left(\frac{6}{14}\right) Entropia(S_{Fuerte})$$

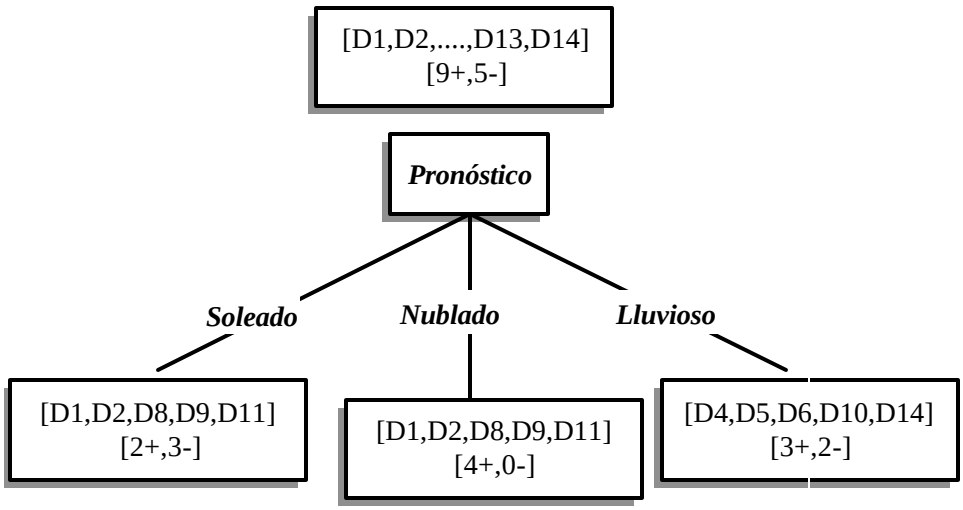
$$= 0.940 - \left(\frac{8}{14}\right) * (0.811) - \left(\frac{6}{14}\right) * (1.00)$$

$$= 0.048$$

C4.5 determina la *Ganancia de Información* para cada atributo candidato (*Pronóstico, Humedad, Temperatura, Viento*), y selecciona el que posea la ganancia más alta. De acuerdo con la medida de ganancia de información, el atributo *Pronóstico* provee la mejor predicción del atributo objetivo (*Jugar-Tenis*), sobre los datos de entrenamiento. *Pronóstico* es seleccionado como el atributo decisión para el nodo raíz del árbol, y las

ramas son creadas debajo de la raíz para cada posible valor del atributo (Soleado, Nublado, Lluvioso).

**Figura 2.3. Construcción del primer nivel de árbol de Decisión**

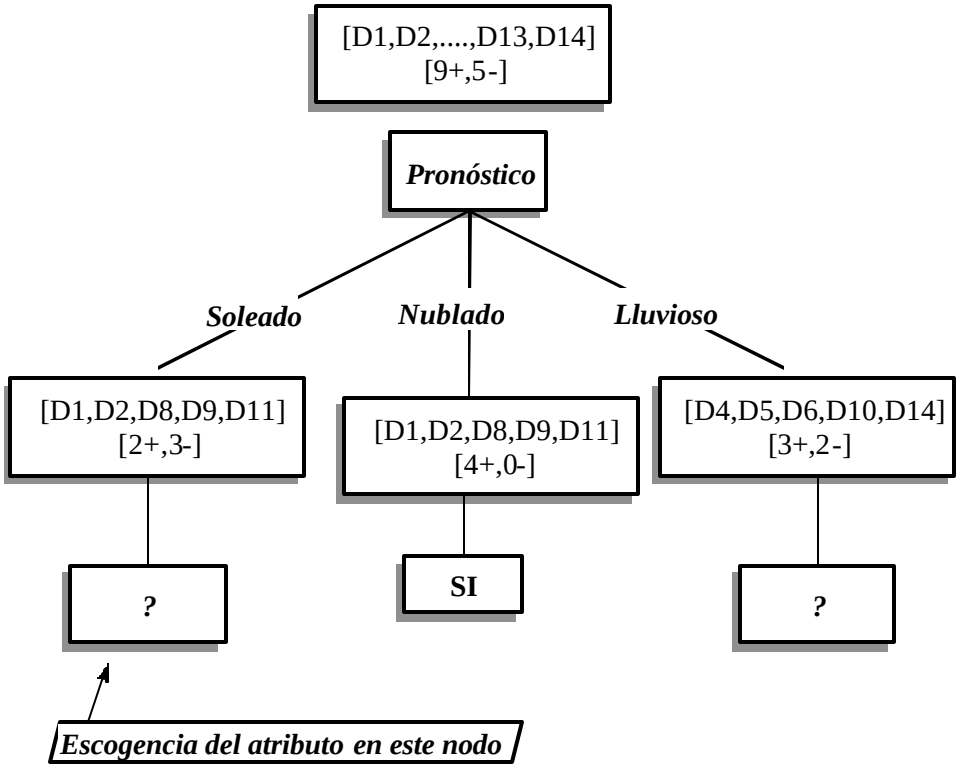



El resultado de la partición del árbol de decisión se muestra en la Figura 2.3, con los datos de entrenamiento ordenados para cada nuevo nodo descendente. Para cada objeto donde Pronóstico es igual a Nublado, se presenta una clasificación *Positiva* sobre la clase decisión, el nodo esta conformado por un nodo hoja en el cual la clasificación es *Jugar-Tenis* = ***Si*** y el valor de la entropía es cero. En contraste con los otros nodos descendientes, *Pronostico = Soleado* y *Pronostico = Lluvioso*, donde el valor de entropía no es cero, y un nuevo árbol debe ser generado por debajo de estos nodos.

El proceso de selección de un nuevo atributo y el particionamiento de los datos de entrenamiento es nuevamente repetido para cada nodo descendiente no terminal, a este nivel solo se usan los datos de entrenamiento asociados con este nodo. Los atributos que son incorporados en la parte de arriba del árbol son excluidos. En el camino desde la raíz del árbol hasta un nodo hoja solo pueden aparecer atributos que no se repitan. Este proceso continúa para cada nuevo nodo hoja, hasta que se cumplan dos condiciones necesarias:

- Cada atributo ha sido incluido a lo largo del camino del árbol, o
- Los datos de entrenamiento asociados con un nodo hoja, tengan todos el mismo valor de atributo (su entropía es cero).

**Figura 2.4. Selección de un nuevo atributo**




En la Figura 2.4 se ilustra el proceso de escogencia de un nuevo nodo, para esto hay que calcular nuevamente la Ganancia de Información para la próxima etapa de crecimiento del árbol de decisión.

Calculamos el valor de Ganancia a la rama *Pronóstico = Soleado*, para crear el nodo y así poder generar el subárbol:

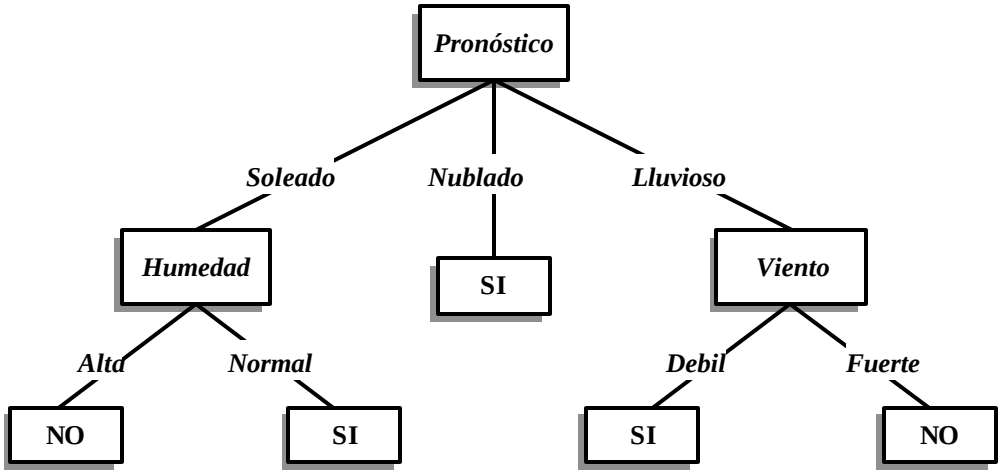
$$Gain(S_{Soleado}, Humedad) = 0.970 - \left(\frac{2}{5}\right)*(0.0) - \left(\frac{3}{5}\right)*(0.0) = 0.970$$

$$Gain(S_{Soleado}, Temperatura) = 0.970 - \left(\frac{2}{5}\right)*(0.0) - \left(\frac{3}{5}\right)*(1.0) - \left(\frac{1}{5}\right)*(0.0) = 0.570$$

$$Gain(S_{Soleado}, Viento) = 0.970 - \left(\frac{2}{5}\right)*(0.0) - \left(\frac{3}{5}\right)*(0.918) = 0.019$$

El atributo con mayor *Ganancia de información* es *Humedad* por lo cual se anexa al árbol y así se debe continuar con la clasificación de los datos de entrenamiento. El árbol construido por completo se muestra en la Figura 2.5.

**Figura 2.5. Árbol de decisión**




Las reglas deducibles a través del árbol de decisión de la Figura 2.5 son:

*1. Pronostico = Soleado Ù Humedad = Alta ® **No***
2. Pronostico = Soleado ( Humedad = Normal ( Si
3. Pronostico = Nublado ( Si
4. Pronostico = Lluvioso ( Viento = Débil ( No
5. Pronostico = Lluvioso ( Viento = Fuerte ( Si

En la Figura 2.6 se muestra la especificación del algoritmo C4.5, sus entradas, procesos y salida:

**Figura 2.6. Algoritmo C4.5**

***C4:5(Datos de Entrenamiento, Clase-Decisión, Atributos)***

*Datos de Entrenamiento*, son los objetos utilizados en la construcción del árbol.

*Clase-Decisión*, es el conjunto de valores utilizados para predecir el árbol.

*Atributos*, es la lista de los atributos condición que pueden ser probados para el aprendizaje del árbol de decisión.

*Retorna:* un árbol de decisión que correctamente clasifica los objetos.

- Crear un nodo raíz para el árbol.
- Si todos los objetos son positivos, retorne un simple nodo raíz del árbol, con etiqueta = **+**.
- Si todos los objetos son negativos, retorne un simple nodo raíz del árbol, con etiqueta = **-**.
- Si los atributos son vacíos, retorne un simple nodo raíz del árbol, con etiqueta = al valor más común de la Clase-Decisión.
- En otro caso iniciamos
  - A ←El atributo de los atributos que mejor clasifique los objetos (*Ganancia de Información*).
  - El atributo decisión para la raíz ←A.
    - Para cada posible valor de $V_i$, de A,
    - Adicione una nueva rama al árbol perteneciente a la raíz, correspondiente a la prueba A = $V_i$.
    - Buscar en los objetos $V_i$ del subconjunto de objetos que tengan valor $V_i$ para A.
    - Si los objetos $V_i$ es vació.
      - Entonces esta nueva rama adicionarla al nodo hoja con etiqueta = al más común valor de la Clase-Decisión.
      - Sino pertenece esta nueva rama adicionada al subárbol.

      ***C4.5(Datos-Entrenamiento, Clase-Decisión, Atributos-{A})***

## 2.3 ARQUITECTURAS DE INTEGRACION DEL PROCESO DCBD CON SGBD

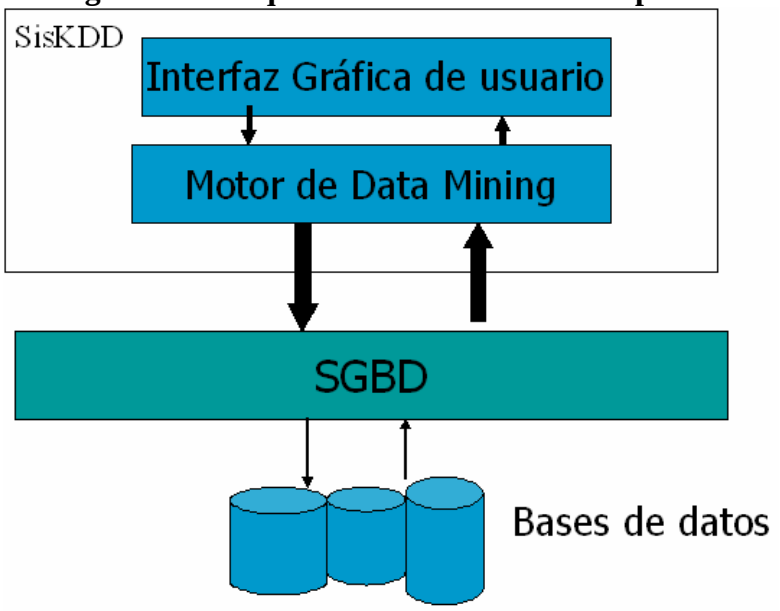
Una herramienta para el descubrimiento de conocimiento en bases de datos debe integrar una variedad de componentes (técnicas de minería de datos, consultas, métodos de visualización, interfaces, etc.), que juntos puedan eficientemente identificar y extraer patrones interesantes y útiles de los datos almacenados en las bases de datos.

Las arquitecturas de estas herramientas se pueden ubicar en una de tres tipos: sistemas débilmente acoplados, medianamente acoplados y fuertemente acoplados con un SGBD.

**2.3.1 Arquitecturas Débilmente Acopladas 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.

**Figura 2.7. Arquitecturas Débilmente Acopladas**

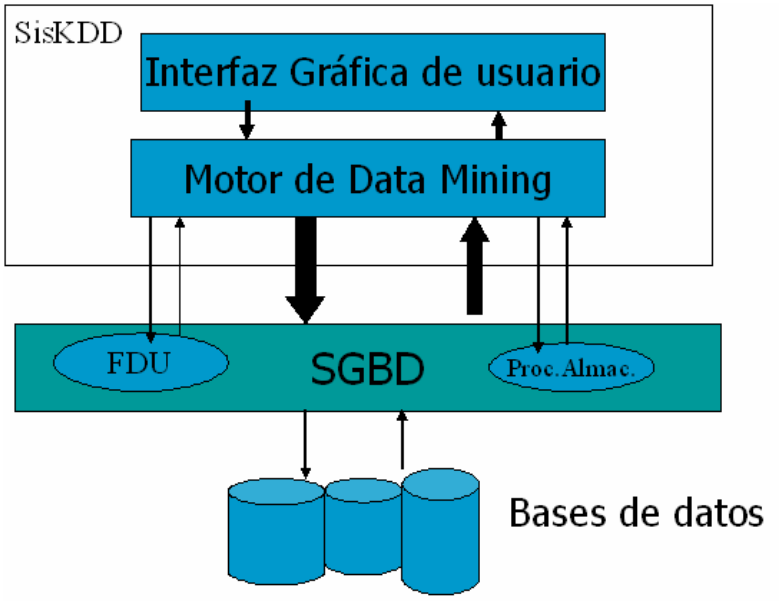



Las principales características de ésta arquitectura son:

- El sistema de Data Mining extrae los datos de la base de datos a través de un ODBC.
- Los datos son leídos tupla por tupla del SGBD hacia el kernel de Data Mining usando un cursor interface.
- El SGBD corre en un espacio de direccionamiento diferente al proceso de Data Mining.
- Consume más memoria y recursos de CPU.
- A pesar de la optimización de lectura por bloques, donde un bloque de tuplas puede ser leído al tiempo, el rendimiento se puede afectar, debido a la carga de comunicaciones entre procesos y conmutaciones de contexto del proceso (disco-memoria).

**2.3.2 Arquitecturas Medianamente Acopladas con un SGBD.** 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.

**Figura 2.8. Arquitecturas Medianamente Acopladas**




Las características de la Integración a través de procedimientos almacenados son:

- Un procedimiento almacenado es un conjunto nombrado de instrucciones y lógica de procedimientos de SQL compilado, verificado y almacenado en la base de datos.
- Los algoritmos de Data Mining son encapsulados como procedimientos almacenados y corren en el mismo espacio de dirección como el SGBD.
- El motor de Data Mining invoca al procedimiento almacenado y le transmite los parámetros requeridos para hacer un trabajo. Un solo mensaje desencadena la ejecución de un conjunto de instrucciones SQL almacenadas.
- Brindan mejor desempeño, ya que ejecutan todas las conexiones, aplicaciones y la base de datos en el mismo espacio de direccionamiento.
- Un procedimiento almacenado recibe igual trato que cualquier otro objeto de base de datos y debe ser registrado en el catálogo.

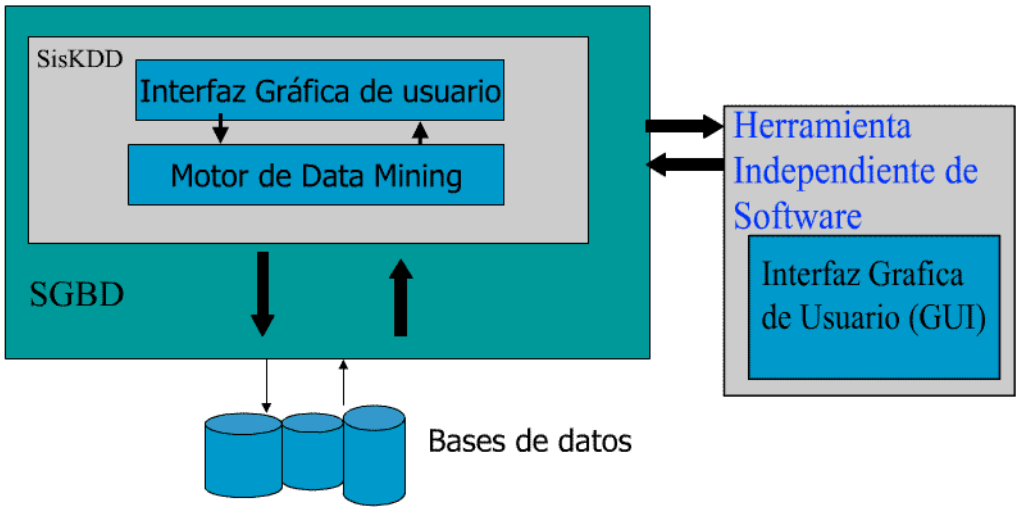
Las características de la Integración a través de funciones definidas por el usuario (UDFs) son:

- Definidas e implementadas por los usuarios en un lenguaje de programación con su descripción, parámetros de entrada, salida y algunos otros atributos.
- El ejecutable de una UDF se almacena en el SGBD.
- El SGBD puede acceder e invocar la función cuando esta sea referenciada en una instrucción SQL.
- Los algoritmos de Data Mining se expresan como una colección de funciones definidas por el usuario (UDFs).

- La mayoría del procesamiento sucede en la UDF y el SGBD se usa principalmente para proveer tuplas hacia las UDFs.
- La principal ventaja sobre los procedimientos almacenados es el desempeño ya que pasar tuplas a un procedimiento almacenado es mas lento que hacia una UDF. Del resto el procesamiento es el mismo.
- La principal desventaja es el costo de desarrollo.

**2.3.3 Arquitecturas Fuertemente Acopladas 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 éste.

**Figura 2.9. Arquitecturas Fuertemente Acopladas**




**2.3.4 Propuestas de Arquitecturas de DCBD Fuertemente Acopladas con un SGBD**
Hay propuestas de investigación que discuten cómo tales sistemas pueden ser implementados y las extensiones del lenguaje SQL necesarias para soportar estas operaciones de descubrimiento de patrones: DMQL, M-SQL, MINE RULE, NonStop SQL/MX y Nuevas Primitivas SQL para Asociación y Clasificación.

En las propuestas de extensión del SQL: DMQL y M-SQL, a pesar de considerar estos lenguajes dentro del grupo de las arquitecturas fuertemente acopladas, por su poder de expresar operaciones Data Mining con una sintaxis SQL, sus implementaciones no lo son. Partes de DMQL están implementadas en el sistema DBMiner, un sistema para Data Mining débilmente acoplado con SGBD relacional. De igual forma, M-SQL hace parte del prototipo Data Mine, un sistema débilmente acoplado para Data Mining que incluye un optimizador de consultas que compila M-SQL queries en un eficiente plan de ejecución por

encima de la base de datos y provee además una interfaz de programación de aplicaciones y lenguaje de consulta para Data Mining.

Rosa Meo et al. propone un modelo unificado para descubrir reglas de asociación, basado en un nuevo operador, llamado MINE RULE, diseñado como una extensión del lenguaje SQL y una semántica formal para este operador. La semántica se describe por medio de una extensión del álgebra relacional con nuevos operadores, que permite transformar una relación con el fin de descubrir reglas de asociación. Además, proponen una arquitectura, que a juicio de los autores es fuertemente acoplada con un SQL server, para soportar el operador MINE RULE.

La desventaja es la arquitectura en la cual se ha implementado el operador MINE RULE que a pesar que los autores consideran fuertemente acoplada con un SGBD, esta no lo es, debido a que los algoritmos de minería de datos forman parte de un módulo independiente llamado kernel por encima del SGBD. En este módulo se realiza todo el proceso de descubrimiento de reglas. El SGBD se utiliza únicamente para almacenar los datos iniciales, las reglas de salida, los resultados intermedios y para realizar ciertas tareas del operador MINE RULE (como evaluación de subconsultas, agrupamientos, etc.), que según los autores, son mas eficientes hacerlas por fuera del core operator.

En [Clear et al. 99a] se reporta la implementación de un conjunto de primitivas, al interior de NonStop SQL/MX, un SGBD Objeto-relacional, paralelo, de la División Tandem de Compaq. El conjunto de primitivas permite soportar, de manera eficiente y escalable, algunas tareas básicas de descubrimiento de conocimiento, sacando provecho y potencia del motor paralelo. Este tipo de integración se enmarca, por lo tanto, en una solución muy específica ya que otros motores no paralelos seguramente no pueden aprovechar estas ventajas.

Timarán [Tima02b] [TiMM03] plantea un nuevo método de acoplar las tareas de Asociación y Clasificación a un SGBD. El método consiste en implementar inicialmente operadores algebraicos al interior del motor del SGBD que faciliten estas tareas y extender el lenguaje SQL con nuevas primitivas que los soporten. Su implementación permitirá contar verdaderamente con una arquitectura de DCBD fuertemente acoplada con un SGBD.

# 3. MANEJADOR DE BASES DE DATOS POSTGRESQL

Este capítulo comprende los tópicos mínimos para conocer la arquitectura, el tratamiento que presenta una consulta y la ejecución de la misma en el SGBD PostgreSQL. Este es el preámbulo al análisis central del presente proyecto.

## 3.3 INTRODUCCION

Un sistema manejador de bases de datos (SMBD), debe proporcionar un entorno que sea eficaz y eficiente para su utilización en la extracción y almacenamiento de información en la base de datos. SMBD se diseñan para gestionar grandes bloques de información, incluyendo tanto la definición de estructuras para su almacenamiento como la provisión de mecanismos para su administración. La importancia de la información en la mayoría de las organizaciones, y por tanto, el valor de los datos almacenados en la base de datos, lleva al desarrollo de una gran cantidad de conceptos y técnicas útiles para la gestión eficiente de los datos.

La extracción de información a partir de bases de datos se debe hacer de forma eficiente, para esto los SMBD disponen de un lenguaje de consulta estructurado *(SQL Structured Query Language)*, que trabaja mediante operaciones básicas, denominadas operadores. Cuando una consulta se lanza, esta se descompone en operaciones que el sistema puede ejecutar. Estos operadores realizan *la selección, búsqueda, proyección,* entre otros, de forma eficiente. El SMBD *PostgreSQL* soporta el modelo de datos relacional y además se extiende al manejo sencillo de *clases, herencia de tipos y funciones*; otras características relacionadas con el poder y la flexibilidad son: *restricciones, triggers, reglas e integridad de transacción.* Así, *PostgreSQL*, se clasifica como un *sistema manejador de base de datos extendido,* es decir, que *PostgreSQL* tenga características de orientado a objeto esta firmemente en el mundo de las bases de datos relacionales. El código de *PostgreSQL* se implementa completamente en ANSI C y al compilarse cumple con los requerimientos de ANSI SQL. Los lenguajes *SQL 92* y *SQL 93* soportan los conceptos de integridad y extensibilidad del modelo Relacional Extendido.

## 3.4 ARQUITECTURA DE POSTGRESQL

Los conceptos presentados a continuación fueron tomados de [FONG86], [LOCK98a], [MOM97], [MOM98], [MOS03], [OJ01], [SIMK98] y [STRW86].

PostgreSQL es un Sistema Gestor de Bases de Datos Objeto-Relacional. Esto quiere decir que PostgreSQL es un sistema de manejo de Bases de Datos relacional (SMBD) que soporta un modelo de datos consistente en una colección de relaciones con nombre, que contienen atributos de un tipo específico. A su vez, tiene algunas características que son propias del mundo de las bases de datos orientadas a objetos como son el soporte a

conceptos tales como:

- Clases
- Herencia
- Tipos
- Funciones

PostgreSQL es una mejora de Postgres. El código original de Postgres fue desarrollado en la Universidad de California, Berkeley; en 1986, por un grupo integrado por estudiantes de pregrado y postgrado de la universidad, y desarrolladores dirigidos por el profesor Michael Stonebraker. Originalmente, Postgres implementó su propio lenguaje de consultas POSTQUEL. La primera versión de Postgres se lanzó en Junio de 1989. En 1994 se añadió un intérprete de lenguaje SQL a Postgres.

En 1995 se liberó el código fuente de Postgres, y se le dio el nombre Postgres95. Postgres95 fue adaptado a ANSI C y se adoptó el lenguaje de consultas SQL, como lenguaje, además se utilizó GNU make para la compilación. En 1996 se cambió el nombre a PostgreSQL y se añadieron características adicionales que cumplen el estándar SQL92.

**3.2.1 Características de Postgres.** Postgres (POSTinGRES) es el sucesor del sistema de base de datos relacionales INGRES, que provee un mejor soporte a aplicaciones de inteligencia artificial y desarrollo de ingeniería que su antecesor. Esta meta es cumplida por Postgres, extendiendo el lenguaje de consulta relacional, incluyendo:

- Soporte a tipos de datos abstractos (Abstract Data Type ADT) y métodos de acceso definidos por el usuario.
- Nuevo lenguaje, QUEL, para soportar operaciones de clausura transitiva requeridas en aplicaciones de Inteligencia Artificial.
- Soporte del lenguaje QUEL como un tipo de dato, para incrementar el poder de modelamiento del sistema relacional.
- Soporte de Reglas y Triggers en sistemas relacionales para mejorar la inferencia y encadenamiento hacia delante, requerido por aplicaciones de sistemas expertos.

El lenguaje de consulta que soporta esta característica es llamado POSTQUEL.

Postgres soporta tipos de datos abstractos, permitiéndole al usuario definir sus propios tipos de datos, para simplificar la representación de información compleja. Además, permite definir operadores para ser utilizados, junto con los tipos de datos definidos por el usuario. Permite a los usuarios extender los métodos de acceso existentes, para ser utilizados con los nuevos operadores, por ejemplo, el usuario puede especificar métodos de acceso para el recorrido eficiente de una relación cuando un operador no estándar aparece en una cláusula de restricciones.

El modelo relacional no es el adecuado para representar relaciones jerárquicas.

Postgres propone Consultas Embebidas dentro de los campos de datos y utilizar estas consultas para expresar la relación jerárquica entre la tupla correspondiente y la información en la base de datos.

**3.2.2 Conceptos de Arquitectura de PostgreSQL.** PostgreSQL utiliza un modelo de arquitectura de proceso por usuario cliente/servidor; por lo cual el Sistema Manejador de Base de Datos de PostgreSQL se ejecuta en un equipo diferente al de las aplicaciones que acceden a las Bases de Datos para proveer protección a los datos. En el establecimiento de la conexión se requiere de la colaboración de los siguientes procesos:

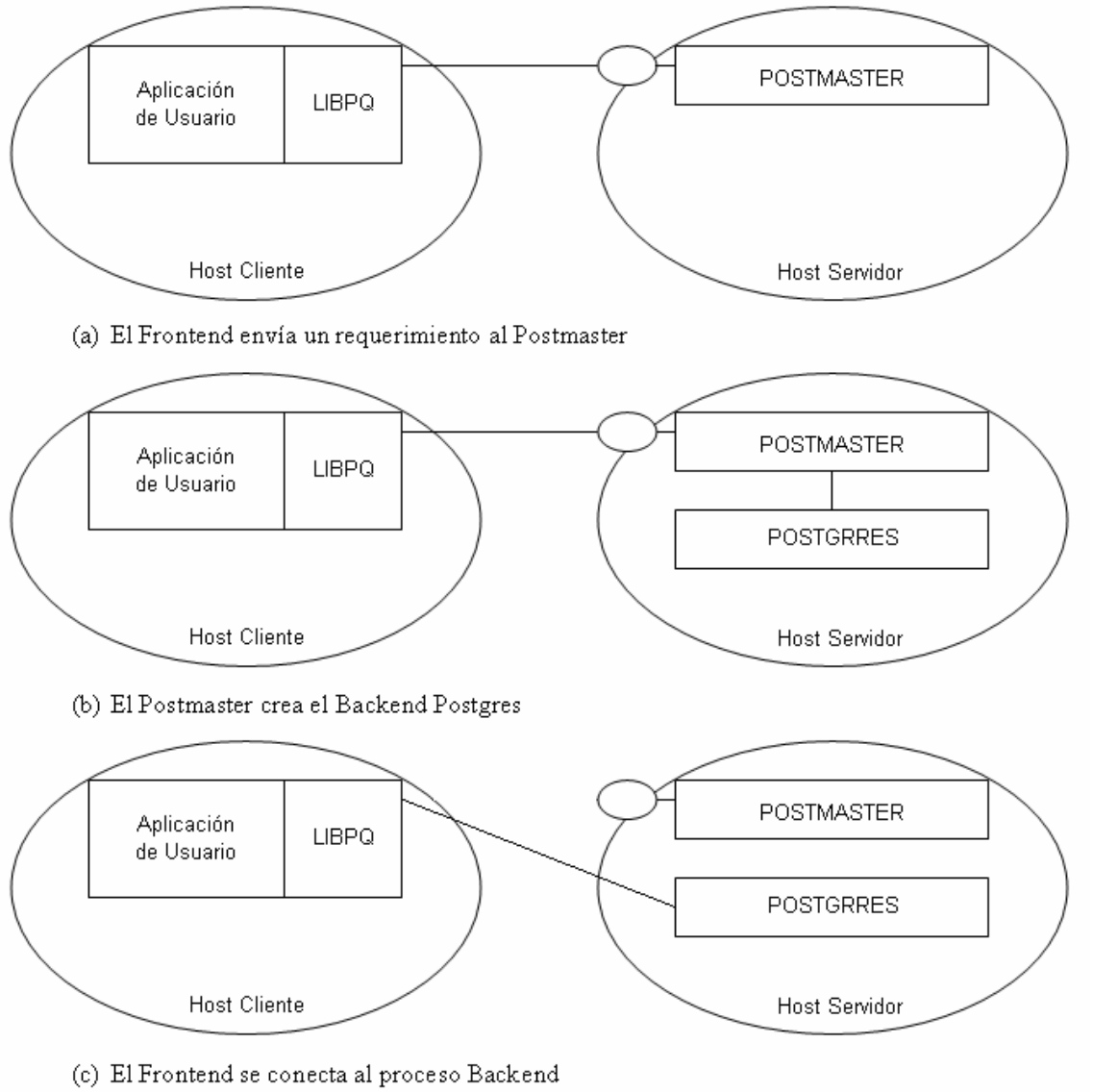
- *Postmaster*: Proceso del demonio supervisor.
- *Fronten*d*:* La aplicación del usuario (programa psql)
- *Backend*: Servidores de bases de datos (mismo Postgres)

El proceso postmaster es el encargado de atender un puerto TCP/IP especifico, a través del cual las aplicaciones frontend, que quieren acceder a una base de datos específica, dentro de una instalación, hacen los llamados respectivos. Cada vez que el proceso postmaster detecta una solicitud de conexión, inicia un proceso de servidor backend y conecta el proceso frontend al nuevo servidor backend. De esta forma, el proceso postmaster queda libre para esperar una nueva solicitud mientras el proceso frontend se comunica con Postgres, a través del servidor backend. El servidor backend, comunica entre si, los procesos Postgres, usando semáforos y memoria compartida para asegurar la integridad de los datos, a través de los accesos concurrentes a la base de datos.

El proceso cliente puede ser un psql frontend (para consultas iterativas SQL) o alguna aplicación implementada utilizando la librería libpq.

Una vez establecida la conexión, el proceso cliente puede enviar una consulta al backend. La consulta es transmitida en texto plano, no se realiza ningún proceso parser en el lado del cliente. El servidor realiza el proceso de parser a la consulta, crea el plan de ejecución, ejecuta el plan y retorna las tuplas al cliente transmitiéndola a través de la conexión establecida.

**Figura 3.1. Proceso de conexión**




## 3.3 PROCESAMIENTO DE UNA CONSULTA EN POSTGRESQL

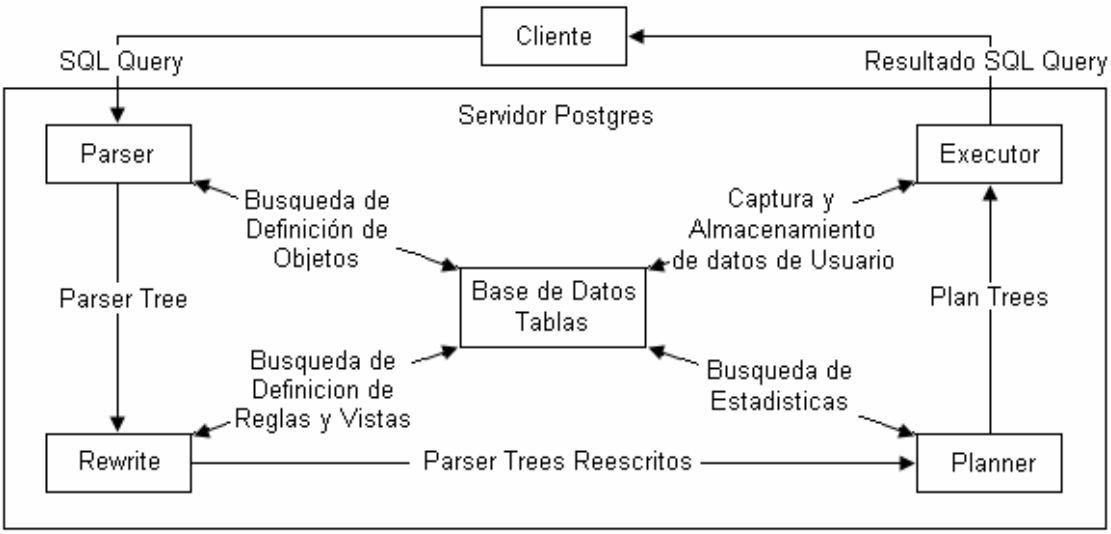
Las etapas que se deben realizar, a gran escala, para procesar una consulta en PostgreSQL son las siguientes:

- Se establece una conexión desde un programa aplicación al servidor Postgres. A través de esta conexión, el programa aplicación transmite las consultas y recibe los resultados del Servidor Postgres. Las consultas son transmitidas como una cadena de texto plano.
- Se realiza un análisis léxico, sintáctico y semántico a la consulta transmitida. Entre las comprobaciones que se realizan, se verifica la existencia de las relaciones y

atributos indicados en la consulta. Si la consulta no presenta errores se crea un árbol de consulta llama query tree. Esta etapa es conocida como etapa Parser.

- En la siguiente etapa se toma la estructura query tree, y se modifican las Reglas o Vistas (VIEW) que se puedan aplicar dentro de la estructura query tree. Esta transformación la realiza el sistema rewrite, sistema de reescritura. Siempre que se presenta una Vista en la consulta el sistema rewrite, transforma la consulta del usuario en una consulta que acceda a las tablas base, dadas en la definición de la vista inicial.
- En la etapa de optimización, se toma la estructura query tree, resultado de la etapa rewrite, y se crean todas las posibles rutas de acceso que conduzcan al mismo resultado. Se selecciona aquella ruta de acceso que tenga el menor costo de ejecución y se crea el plan de la consulta, dentro de la estructura query plan.
- El encargado de ejecutar la estructura query plan y de recuperar las tuplas indicadas en la consulta es el executor. En esta etapa, se toma la estructura query plan y se ejecuta recursivamente. Se buscan relaciones, se realizan ordenamientos y Joins, se evalúan cualificaciones, se hace uso del sistema de almacenamiento y se recuperan las tuplas en la forma representada en el plan.

**Figura 3.2. Procesamiento de una consulta**




**3.3.1 Etapa Parser.** La etapa parser consiste de dos procesos: el proceso parser y el proceso de transformación.

**3.3.1.1 Proceso Parser.** El parser verifica la validez sintáctica de la cadena de consulta. Si la sintaxis es correcta se construye y se retorna una estructura parser tree, de lo contrario, se retorna un mensaje de error. Para implementar esta etapa se usa las herramientas Lex y Yacc (Revisar Anexo C).

El análisis léxico se define en el archivo scan.l y su objetivo es reconocer los identificadores, palabras reservadas SQL, etc. Para cada identificador o palabra clave identificada, un token se genera y se envía al parser (…/src/backend/parser/scan.l).

El análisis semántico se define en el archivo gram.y y consiste en un conjunto de reglas gramaticales y acciones que son ejecutadas cuando una regla se dispara. El código de las acciones está definido en sentencias de código C y se utiliza para construir una estructura parse tree (…/src/backend/parser/gram.y).

El archivo scan.l se transforma a un archivo en código C llamado scan.c, usando el programa lex y el archivo gram.y se transforma a un archivo en código gram.c utilizando la herramienta yacc. Después de realizadas estas transformaciones, un compilador normal de C se puede usar para crear el parser. Note que las transformaciones y compilaciones mencionadas se realizan a través de archivos Makefile distribuidos en el código de PostgreSQL.

Para entender mejor la estructura de datos usada en PostgreSQL para el procesamiento de una consulta, se ilustrará con un ejemplo los cambios hechos a la estructura de datos en cada etapa.

Ejemplo 3.1. Este ejemplo contiene la siguiente consulta simple que será usada en varias descripciones y figuras a través del ejemplo de procesamiento de una consulta en PostgreSQL. La consulta asume que las tablas ya están definidas.

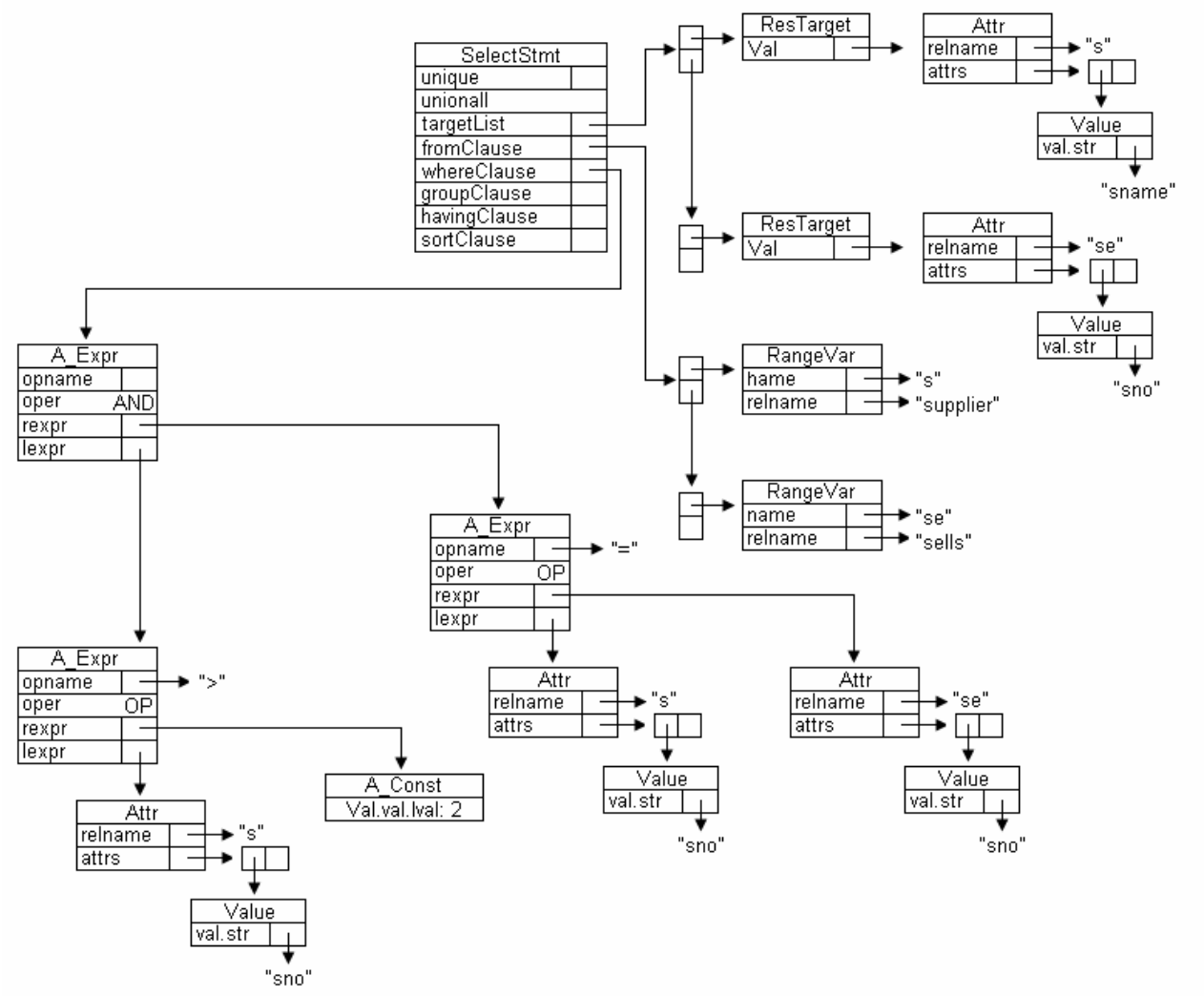
```
select s.sname, se.pno
from supplier s, sells se
where s.sno > 2 and
        s.sno = se.son;
```

La Figura 3.3 muestra el parser tree construido por las reglas gramaticales dadas en gram.y para la consulta dada en el ejemplo 3.1.

El nodo raíz del árbol es un nodo SelectStmt. Para cada entrada que aparezca en la cláusula FROM de la consulta SQL, un nodo RangeVar se crea conteniendo el nombre del alias y el nombre de la relación. Todos los nodos RangeVar se coleccionan en una lista que es añadida al campo fromClause del nodo SelectStmt.

Para cada entrada que aparece en la lista SELECT de la consulta SQL, se crea un nodo ResTarget, que contiene un apuntador a un nodo Attr. El nodo Attr contiene el nombre de la relación de entrada y un puntero a un nodo Value que contiene el nombre del atributo. Todos los nodos ResTarget son coleccionados en una lista que está conectada al campo targetList del nodo SelectStmt.

**Figura 3.3. Parser tree de la consulta del ejemplo 3.1**



El operador tree se construye para la cláusula WHERE de la consulta SQL, la cual es añadida al campo qual del nodo SelectStmt. En la Figura 3.3 la raíz del operador tree es un nodo A_Expr que representa una operación AND. Este nodo tiene dos sucesores llamados lexpr y rexpr apuntando a dos subárboles. El subárbol apuntado por lexpr representa la cualificación s.son > 2 y el subárbol que apunta al rexpr representa la cualificación s.son = se.son. Para cada atributo, se crea un nodo Attr conteniendo el nombre de la relación y un puntero al nodo Value que contiene el nombre del atributo. Para las constantes que aparecen en la consulta, se crea un nodo Const es creado conteniendo el valor de la constante.

**3.3.1.2 Proceso de Transformación.** El proceso de transformación toma el nodo tree retornado por la etapa parser como estructura de entrada y lo recorre recursivamente. Si un nodo SelectStmt se encuentra, éste se transforma en un nodo Query, que será el nodo raíz de la nueva estructura de datos. La Figura 3.4 muestra la estructura de datos transformada.

Se realiza una verificación de la existencia en el sistema, sobre las relaciones mencionadas en la cláusula From. Para cada relación que está presente en el catálogo, un nodo RTE se crea, conteniendo el nombre de la relación, el nombre alias y el id de la relación. De aquí en adelante los id's de la relación se usan para referenciar las relaciones dadas en la consulta. Todos los nodos RTE se coleccionan en la lista range table entry list (Lista de entrada de tablas de rango) que esta conectada el campo rtable del nodo Query. Si un nombre de relación no es conocido por el sistema en la consulta, un error se retorna y el procesamiento de la consulta será abortado.

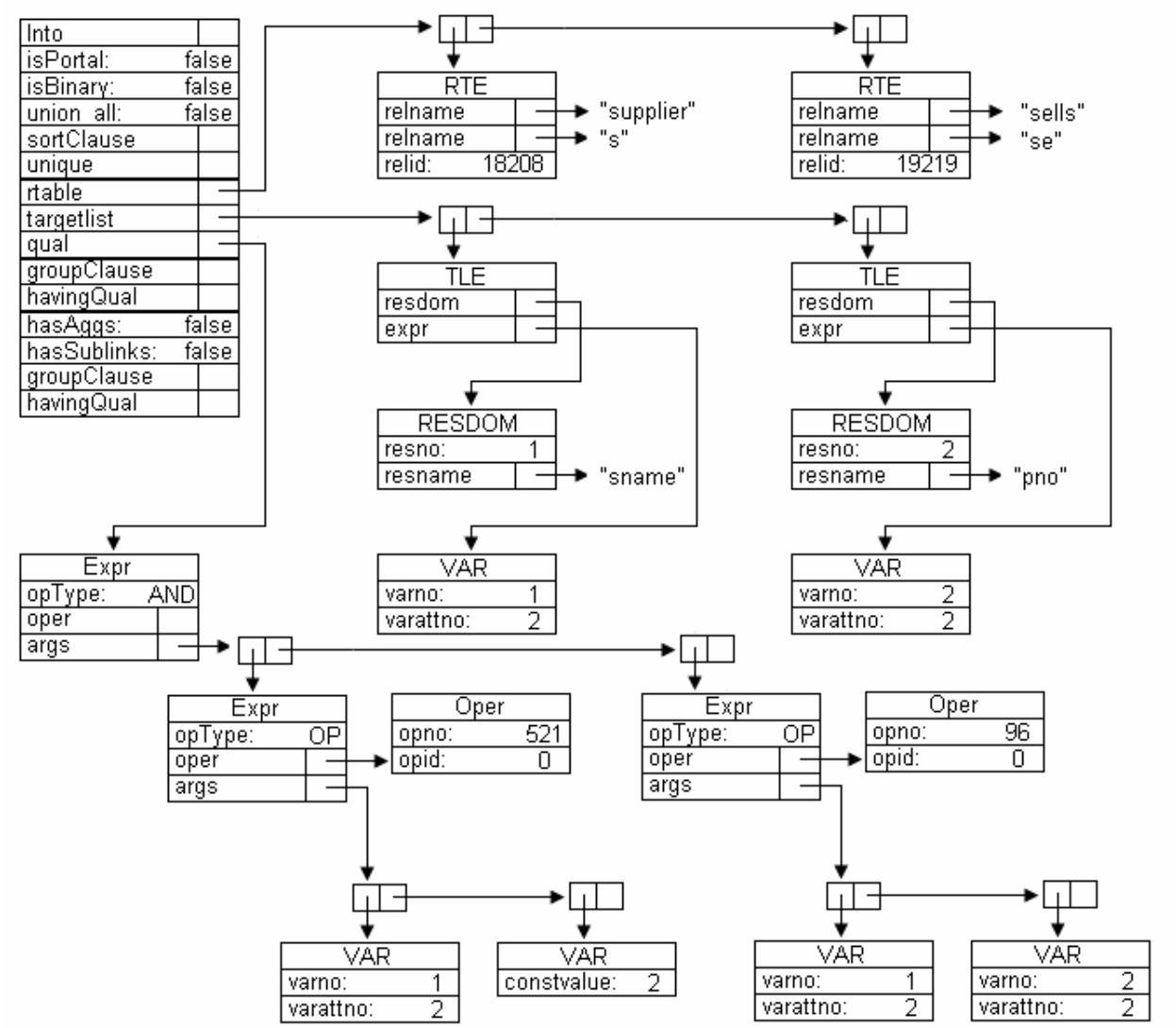
Una vez se verifican los nombres de las relaciones, se procede a verificar si los nombres de los atributos usados en la consulta, están contenidos en las relaciones dadas. Para cada atributo, se crea un nodo TLE, conteniendo un puntero a un nodo Resdom (el cual contiene el nombre de la columna) y un puntero a un nodo Var. En el nodo Var hay dos números importantes. El campo varno dá la posición de la relación que contiene el atributo actual en el range table entry list. El campo varattno dá la posición del atributo dentro de la relación. Si el nombre de un atributo no se encuentra, se retorna un error y el procesamiento de la consulta se aborta.

Cada nodo A_Expr se transforma en un nodo Expr. El nodo Attr que representa los atributos, se reemplaza por nodos Var, como se realizó en el campo targetlist. Se realiza una verificación si los atributos que aparecen son válidos y conocidos por el sistema. Si algún atributo no es conocido por el sistema, se retorna error y el procesamiento de la consulta se aborta.

El proceso de transformación, transforma la estructura de datos retornada por la etapa parser, para obtener una estructura de mayor manipulación para las etapas restantes del proceso de optimización. Las cadenas de caracteres representando relaciones y atributos en el árbol original se reemplazan por los id's de la relación y nodos Var cuyos campos se refieren a las entradas en el range table entry list. Además en la etapa de transformación,

también se realizan verificaciones sobre la validez de los nombres de las relaciones y atributos en el contexto actual.

**Figura 3.4. Parser tree transformado, de la consulta del ejemplo 3.1**



**3.3.2 Sistema Rewrite.** PostgreSQL soporta un poderoso sistema de reglas para la especificación de vistas y vistas actualizables. El sistema de reglas de PostgreSQL consiste en el sistema Rewrite. El sistema query rewrite es un módulo entre la etapa parser y la etapa del optimizador de consultas. En esta etapa se toma el árbol devuelto por la etapa parser y se realiza la búsqueda de reglas presentes dentro de la consulta. Si se encuentra alguna regla dentro de la consulta, la estructura de árbol se transforma en una expresión equivalente que incluye las relaciones base.

La reescritura de la consulta se realiza en el archivo …/src/backend/rewrite/rewriteHandler.c.

**3.3.3 Etapa Planner/Optimizer.** La Tarea de la etapa de optimización es crear un plan de ejecución óptimo. Primero, se combinan todas las posibles formas de recorrer y unir las relaciones que aparecen en la consulta. Todos los Path's creados llevan al mismo resultado y la tarea del optimizador es estimar el costo de ejecución de cada Path y encontrar cúal de éstos es el más barato. La generación del plan de ejecución se realiza en el archivo …/src/backend/optimizar/plan/planner.c.

El optimizador decide cuales planes se deben generar, basado en los tipos de índices definidos en las relaciones que aparecen en la consulta. Un plan usando escaneo secuencial siempre se crea. Si un índice es definido para una relación y una consulta contiene una restricción de la forma *relación.atributo* OPR *constante* y *relación.atributo* coincide con la clave del índice y OPR es un operador relacional distinto del operador ?, se crea otro plan utilizando el índice para recorrer la relación. Si existen otros índices y en las restricciones de la consulta coinciden con las claves de dichos índices, estos serán considerados por el optimizador.

Después de que todos los planes factibles para recorrer una relación simple han sido encontrados, se crean los planes para unir las relaciones. El optimizador considera solamente Joins entre dos relaciones, para las cuales existe una correspondiente cláusula Join (por ejemplo una restricción como where rel1.attr1 = rel2.attr2) en la cualificación de la cláusula WHERE. Todos los posibles planes son considerados para cada pareja de JOIN tomadas en cuenta por el optimizador.

Las posibles estrategias JOIN consideradas por el optimizador son:

- Join nested: La relación de la derecha se recorre una vez por cada tupla encontrada en la relación de la izquierda. Esta estrategia es fácil de implementar pero puede consumir mucho tiempo.
- Join merge-sort: Cada relación se ordena por los atributos del Join antes de iniciar el Join. Después las dos relaciones se mezclan, tomando en cuenta que ambas relaciones están ordenadas sobre los atributos del Join. Esta clase de Join es más atractiva porque cada relación se recorre una sola vez.
- Join Hash: La relación de la derecha es primero hashed en los atributos del Join. Después la relación de la izquierda se recorre y los valores apropiados de cada tupla

encontrada son usados con clave hash para ubicar las tuplas en la relación de la derecha.

La Figura 3.5 muestra el plan producido por la consulta en el ejemplo 3.1.

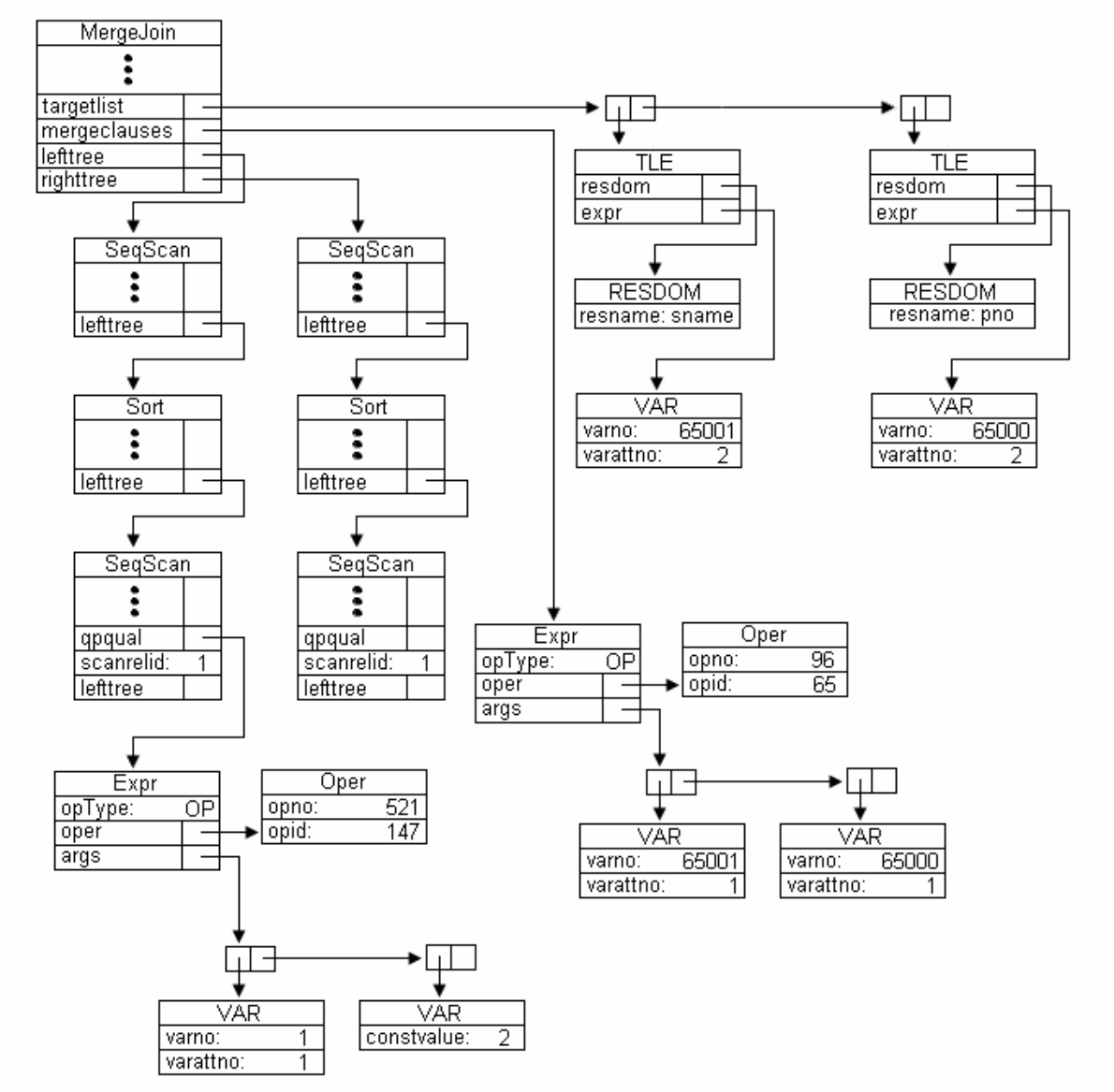
A continuación se da una breve descripción de los nodos que aparecen en el plan.

El nodo raíz del Plan, es un nodo MergeJoin con 2 sucesores, uno atado al campo lefttree y el segundo atado al campo righttree. Cada uno de los subnodos representa una relación Join. Un Join merge-sort requiere que cada relación se encuentre ordenada. Por esto, se encuentra un nodo Sort en cada SubPlan. La cualificación adicional dada en la consulta *s.sno > 2* se desplaza hacia abajo, lo mas lejos que sea posible y se enlaza al campo qpqual del nodo SeqScan del correspondiente subplan.

La lista enlazada al campo mergeclauses del nodo MergeJoin contiene información acerca de los atributos del Join. Los valores 65000 y 65001 para los campos varno en el nodo Var, que aparecen en la lista mergeclauses (está contenida en el campo targelist), significa que no se debe considerar las tuplas del nodo actual sino las tuplas del siguiente nodo en profundidad, por ejemplo los nodos raíces del subplan.

Otra tarea realizada por el optimizador es establecer el campo operator id en los nodos Expr y Oper. Debido a que PostgreSQL soporta una gran variedad de tipos de datos diferentes y también los definidos por el usuario, es necesario almacenarlas en un sistema de tablas, para poder mantener la enorme cantidad de funciones y operadores. Cada función y operador tiene un único operator id. De acuerdo al tipo de los atributos usados dentro de las cualificaciones se usa el operador apropiado.

**Figura 3.5. Plan para la consulta del ejemplo 3.1**



**3.3.3.1 Estimación de costo y tamaño.** Para podar el espacio de planes, el optimizador debe estimar el costo de ejecución de un plan generado. El optimizador, para estimar el costo, toma en cuenta el tiempo requerido en operaciones CPU y de Entrada/Salida. Cada factor de costo, tiene la siguiente forma:

*Costo = P + W*T.*

donde P es el número de páginas examinadas en tiempo de ejecución por el plan y T es el número de tuplas examinadas. P refleja el costo de operaciones de Entrada/Salida, T refleja el costo de operaciones de CPU y W es un factor de peso que indica la importancia relativa del costo de procesamiento en términos de operaciones de CPU y de Entrada/Salida.

Para un recorrido secuencial de una relación, cada página y tupla se debe examinar, P es el número de páginas en la relación. Para un índice secundario, el costo depende del número de páginas y tuplas en el índice de la relación, porque las páginas y tuplas del índice se deben leer primero para determinar dónde examinar la relación principal.

Para cada índice, el número de páginas y tuplas está determinado por la fracción de tuplas en la relación que se espera satisfagan la cláusula de restricción. Esta estimación recibe el nombre de selectividad. La selectividad está en función de una variedad de parámetros, incluyendo el operador de la cláusula de restricción, la constante de restricción, el número de registros de un índice y los valores máximo y mínimo almacenados en el atributo.

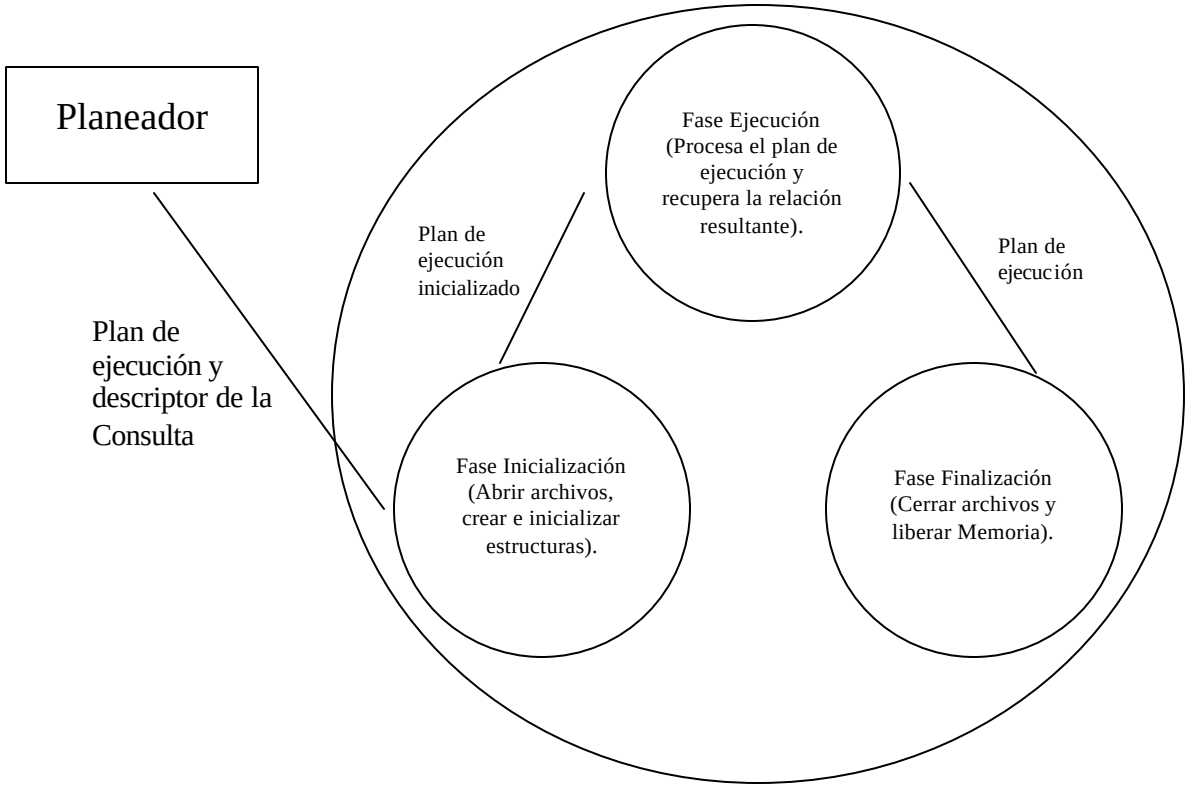
Las fórmulas para la estimación de costo de todas las estrategias Join son funciones del tamaño, en número de páginas y tuplas, de las relaciones del Outer e Inner Join. Para estimar el tamaño en la relación Outer o Inner del Join, el optimizador multiplica el tamaño original de la relación por la selectividad de cada cláusula de restricción aplicable a la relación. Si la cláusula utiliza un índice, la selectividad se calcula, como se indicó anteriormente. Si esto no es posible, el optimizador asocia constantes a estos factores. Si la relación del Outer Join es una relación compuesta, el factor de selectividad está dado por la operación Join. Este factor de selectividad indica la fracción del producto entre las relaciones del Outer e Inner Join, que se espera satisfagan la cláusula Join.

PostgreSQL almacena en el catálogo del sistema, información estadística requerida por las fórmulas de costo y actualiza información del número de páginas y tuplas en una relación, utilizando demonios que se ejecutan en modo background. Estos demonios son implementados utilizando triggers. El valor máximo y mínimo de un atributo, se almacena en la forma de consulta embebida dentro del dato. Esta consulta captura la información apropiada de la base de datos, y el sistema almacena los datos en cache, así el optimizador no necesita ejecutar repetidamente la consulta.

**3.3.4 Etapa Executor.** Dada una consulta, y luego del análisis sintáctico y semántico de ésta, el planeador de PostgreSQL, apoyado en el optimizador, define un plan de ejecución. Este plan representa lógicamente los pasos necesarios para satisfacer dicha consulta, y físicamente es un árbol cuyos nodos representan operaciones básicas.

Una vez creado el plan de ejecución por parte del planeador, este se ejecuta, es decir, se ejecuta la consulta. Para esto, el ejecutor de consultas de *PostgreSQL* se divide en tres fases: inicialización, ejecución y finalización (Figura 3.6).

**Figura 3.6. Fases del Ejecutor de PostgreSQL**




**3.3.4.1 Fase de inicialización.** En esta fase se asignan los recursos necesarios para que el plan de ejecución se lleve a cabo. Básicamente se crean e inicializan estructuras asociadas a los nodos que componen el plan de ejecución, y se abren los archivos correspondientes a las relaciones (tablas) que intervienen en la consulta.

La rutina encargada de realizar la fase de inicialización es *ExecutorStart*, declarada en el archivo ExecMain.c (…/src/backend/executor/ExecMain.c). Esta tiene como parámetros la descripción de la consulta y el estado de ejecución. La descripción de la consulta contiene el árbol semántico, el plan de ejecución y la acción a realizar (select, update, insert, delete). El estado de ejecución contiene información acerca de la dirección de búsqueda, conjunto de relaciones que intervienen en la consulta y bloques de memoria disponibles, entre otros. Esta función retorna la descripción de los atributos del resultado de la consulta, es decir, el descriptor de la relación resultante.

Al interior de *ExecutorStar* se llama a la función *InitPIan,* declarada en el archivo ExecMain.c, la cual en realidad, inicializa el plan de consulta. Esta función recibe los mismos argumentos que *ExecutorStart,* y también retorna el descriptor de la relación resultante.

La función *InitPIan* invoca a *ExecInitNode,* declarada en el archivo ExecProcNode.c (…/src/backend/executor/ExecProcNode.c), con el nodo raíz del plan de ejecución y el estado de la consulta como argumentos. La función *ExecInitNode* es un gran *switch,* que dependiendo del tipo de nodo llama a la función encargada de inicializarlo. Dado el caso que un nodo tenga subplanes de entrada y salida, la función también se usa para inicializarlos. De esta forma se inicializa recursivamente todo el plan de ejecución. La rutina *ExecInitNode* retorna verdadero o falso según se inicialice ó no correctamente el nodo.

PostgreSQL clasifica los nodos (Operadores):

- *Control:* Result, Append
- *Búsqueda:* SeqScan, IndexScan
- *Unión:* NestLoop, MergeJoin
- *Materialización:* Material, Sort, Unique, Group, Agg, Hash, HashJoin

**3.3.4.2 Fase de Ejecución.** Esta es la fase principal, debido a que realiza la ejecución real de la consulta. La función *ExecRun* declarada en el archivo ExecMain.c (…/src/backend/executor/ExecMain.c), es la encargada de realizar esta fase.

La función *ExecutePlan,* declarada en el archivo ExecMain.c, se utiliza por la función *ExecutorRun* para realizar su tarea. La función *ExecutePlan* procesa en la dirección especificada el plan de consulta, y retorna tanto el número de tuplas como las tuplas mismas.

La función *ExecutePlan* invoca a *ExecProcNode,* ubicada en el archivo ExecProcNode.c, con el nodo raíz del plan de ejecución como argumento. La función *ExecProcNode,* al igual que *ExecInitNode,* es un *switch* que identifica el tipo de nodo e invoca a la rutina correspondiente. De esta forma, se obtiene la tupla resultante del proceso realizado al interior del nodo. Esta función también se utiliza para ejecutar los subplanes de entrada y salida de un nodo, y es así como se ejecuta recursivamente todo el plan de ejecución.

**3.3.4.3 Fase de Finalización.** Esta fase corresponde al final de la ejecución de cualquier consulta, y se encarga de liberar todos los recursos que se reservaron en la fase de inicialización. La rutina asociada a esta tarea es *ExecutorEnd,* declarada en el archivo ExecMain.c. Esta rutina tiene los mismos argumentos que *ExecutorStart* y no retorna nada. Al interior de *ExexutorEnd* se llama a la función *EndPlan,* declarada en el archivo ExecMain.c. La función *EndPlan* finaliza el plan de consulta (cerrar archivos y liberar

memoria de estructuras), recibiendo como parámetros el plan y el estado de ejecución. Esta función no retorna nada.

La función *EndPlan* llama a la función *ExecEndNode*, declarada en el archivo ExecProcNode.c. La función *ExecEndNode* recibe los mismos argumentos que la función *EndPlan*, y es también un *switch* que dependiendo del tipo de nodo invoca a la función encargada de finalizarlo. De esta manera, esta función finaliza cualquier tipo de nodo. La función también se utiliza para finalizar los subplanes de entrada y salida de un nodo, y así finalizar recursivamente todo el plan de ejecución con solo finalizar el nodo raíz. La función *ExecEndNode* solo es llamada si el plan ha sido ejecutado sin errores, ya que después de esta operación, el plan de consulta no puede ser ejecutado nuevamente.

Para conocer con más detalle las funciones, estructuras y bibliotecas más importantes de PostgreSQL de la versión 7.3.4, remítase a los anexos A y B.

# 4. OPERADORES ALGEBRAICOS Y PRIMITIVAS SQL PARA EL DESCUBRIMIENTO DE CONOCIMIENTO EN BASES DE DATOS

En este capítulo se da a conocer los operadores y primitivas propuestos en [Tima02b] que se implantaron dentro del SGBD PostgreSQL para soportar eficientemente tareas de minería de datos (asociación y clasificación). Los conceptos presentados fueron tomados de [Tima01] y [TiMM03].

## 4.3 INTRODUCCIÓN

La mayoría de sistemas de minería de datos tienen una arquitectura débilmente acoplada con un SGBD. En esta arquitectura, los costosos algoritmos de descubrimiento de conocimiento y demás componentes, se implementan por fuera del núcleo del SGBD, en una capa externa, y su integración con este se hace a partir de una interfaz cuya función, en la mayoría de los casos se limita a los comandos "leer de" y "escribir en". Esto conlleva a que sean ineficientes en términos computacionales. Por otro lado, un sistema de minería de datos con una arquitectura fuertemente acoplada con un SGBD resuelve los problemas de escalabilidad y rendimiento de los anteriores ya que los algoritmos de minería se encuentran integrados al motor del SGBD, como una operación primitiva y se ejecutan directamente sobre la base de datos.

Para integrar una tarea de minería de datos como una nueva primitiva al motor de un SGBD, es necesario primero determinar si con el conjunto de operadores del álgebra relacional, es posible expresar eficientemente la tarea. Si no es posible, una alternativa es extender el álgebra relacional con nuevos operadores que la soporten y finalmente implementarlos en SQL. Para garantizar la eficiencia de la primitiva, los nuevos operadores algebraicos deberían realizar los procesos computacionalmente más costosos, generados por la tarea de minería de datos.

Es por eso que se proponen nuevos operadores del álgebra relacional que soporten las nuevas primitivas con las que se extiende el lenguaje SQL para expresar eficientemente al interior de un SGBD, tareas de asociación y clasificación.

Un análisis previo [Tima01] sobre el poder expresivo del conjunto de operadores del álgebra relacional para soportar las tareas de asociación y clasificación, mostró que el conjunto de operadores es completo en el sentido que permite expresar tales tareas con los operadores del álgebra relacional. Sin embargo y teniendo en cuenta, que estos operadores han sido implementados en el lenguaje SQL estándar, el proceso de descubrir reglas en grandes volúmenes de datos no es eficiente.

En 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. El algoritmo C4.5 utiliza como métrica la ganancia de información para seleccionar el atributo candidato en cada etapa de construcción del árbol. Se propone un nuevo operador algebraico para clasificación que facilite el cálculo de la ganancia de información ordenando los atributos relevantes, desde la raíz a los nodos terminales, con respecto a su poder de clasificación, i.e. la capacidad para reducir la entropía.

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. Un nuevo operador algebraico para asociación debe facilitar este proceso.

Teniendo en cuenta estos criterios, se ha definido el operador algebraico *Mate* y las funciones agregadas *Gain()* y *Entro()* para clasificación. *Mate* empareja en cada partición todos los atributos condición con el atributo clase, lo que facilita su conteo y el posterior cálculo de las medidas de entropía y ganancia de información. El operador *Mate* genera una nueva relación, en una sola pasada sobre la tabla de entrenamiento (lo que redunda en la eficiencia del proceso de construcción del árbol de decisión), cuyo esquema esta formado por los atributos condición y el atributo clase. Las funciones agregadas *Gain()* y *Entro()* calculan, en cada partición y para cada atributo, la ganancia de información y la entropía, respectivamente. Estas nuevas funciones agregadas se ejecutan conjuntamente con el operador algebraico *Mate.*

Para asociación, se proponen los operadores algebraicos *Associator, Assorow, Assocol y EquiKeep.* A partir de una tupla, *Associator* genera subconjuntos de combinaciones de los valores de los atributos de esa tupla de tamaño variable. A pesar de que generar todas las posibles combinaciones de los valores por cada tupla de una tabla es un proceso computacionalmente costoso, *Associator* lo hace en una sola pasada sobre la tabla. El operador *Associator* facilita el cálculo del soporte de los itemsets candidatos y frecuentes para el algoritmo Apriori y el conteo de los itemsets en FP-tree.

El operador Assorow realiza la misma función que Associator pero necesita que los atributos de la relación sean del mismo tipo con el fin de no hacer nulos los atributos que no intervienen en la formación de un determinado subconjunto. Assorow es útil específicamente en análisis de canasta de mercado.

El operador Assocol genera itemsets de tamaño variable cuando se aplica a lo que Rajamani denomina una relación de modelo simple SC propias del análisis de canasta de mercado.

El operador *EquiKeep,* conserva en cada tupla de una relación los valores de los atributos que cumplen una condición determinada, útil en los algoritmos Apriori y FP-tree, para dejar en cada registro de la tabla los ítems o itemsets frecuentes, una vez hallados por ejemplo los itemsets frecuentes tamaño k-1.

Se proponen también, un operador algebraico auxiliar que se puede utilizar en la etapa de transformación. *Enhance* convierte una relación de modelo simple SC (i.e. una relación con esquema (TID, ITEM)), muy común en análisis de canasta de mercado, en una relación transaccional o multicolumna MC (i.e. una relación con esquema (TID, ITEM1, ITEM2,...ITEMn)).

A continuación se amplia la información de los operadores y primitivas implantadas dentro del SGBD Postgresql contenidos dentro de este trabajo de investigación que se enfoca en la implantación de Associator y EquiKeep por asociación y Mate con sus funciones agregadas Gain() y Entro() para clasificación.

## 4.4 CONCEPTOS PRELIMINARES

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

Formalmente, sea I = {i1, i2,...,im} un conjunto de literales, llamados ítems. Sea D un conjunto de transacciones, donde cada transacción T es un conjunto de ítems tal que $T \subseteq I$. Cada transacción se asocia con un identificador, llamado TID. Sea X un conjunto de ítems. 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 ítems 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 y papas fritas también contienen pañales; el 2% de todas las transacciones los contienen a ambos ítems”. 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). 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.

**4.2.1.1 Tipos de reglas de Asociación.** Según Han Jiawei, existen varios criterios para clasificar las Reglas de Asociación. Uno de ellos, es el de las dimensiones que estas abarcan. De acuerdo con este criterio las reglas de asociación pueden ser unidimensionales y multidimensionales.

Una regla de Asociación es unidimensional si los ítems o atributos de la regla hacen referencia a un solo predicado o dimensión. Por ejemplo la regla de asociación:

Cerveza ^ Papas fritas => pañales

Que puede ser reescrita como:

compra (cerveza) ^ compra (papas fritas) => compra (pañales)

Hace referencia solo a una dimensión: COMPRA.

Una regla de Asociación es multidimensional si los ítems o atributos de la regla hacen referencia a dos o más criterios o dimensiónes. Por ejemplo la regla de asociación:

Edad (30...39) ^ Ocupación (Ingeniero) => compra (laptop)

Contiene tres predicados: EDAD, OCUPACION, COMPRA.

**4.2.2 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. 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 identidad de 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 y se escoge randómicamente del total de tuplas de la base de datos. A cada tupla de este conjunto se denomina ejemplo de entrenamiento. En el segundo paso, se usa el modelo para clasificar. Inicialmente, se estima la exactitud del modelo utilizando otro conjunto de tuplas de la base de datos, cuya clase es conocida, denominado conjunto de prueba. Este conjunto es escogido randómicamente y es independiente del conjunto de entrenamiento. 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, el modelo puede ser usado para clasificar futuros datos o tuplas para los cuales no se conoce a que clase pertenecen.

El modelo de clasificación basado en árboles de decisión es un método de aprendizaje supervisado que construye árboles de decisión a partir de un conjunto de casos o ejemplos (training set) extraidos de la base de datos. También se escoge un conjunto de prueba, cuyas características son conocidas, con el fin de evaluar el árbol. Un árbol de decisión es una estructura donde:

- 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.

Uno de los algoritmos de clasificación con árboles de decisión más utilizados es el C4.5. Este algoritmo construye el árbol de decisión de arriba hacia abajo de forma recursiva. Se trata de construir el árbol de decisión más simple que sea consistente con el conjunto de entrenamiento *T*. Para ello hay que ordenar los atributos relevantes, desde la raíz a los nodos terminales, de mayor a menor poder de clasificación. El poder de clasificación de un atributo *A* 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. El poder de clasificación de un atributo se mide de acuerdo a su capacidad para reducir la incertidumbre o entropía (grado de desorden de un sistema). Esta métrica se denomina Ganancia de información. El atributo con la más alta ganancia de información se escoge como el atributo que forme un nodo en el árbol.

La ganancia de información obtenida por el particionamiento del conjunto T, de acuerdo con el atributo A se define como:

$$Gain(T, A) = I(T) - E(A)$$

donde, I(T) es la entropía del conjunto T, el cual esta compuesto por *s* ejemplos y *m* distintas clases Ci (i=1,m) y se calcula:

$$I(T) = - \sum_{i=1}^{m} pi \log2 (pi)$$

donde, pi = si/s es la probabilidad que un ejemplo cualquiera pertenezca a una clase Ci y *si* es el número de ejemplos de T de la clase Ci.

E(A) es la entropía del conjunto *T* si es particionado por los *n* diferentes valores del atributo *A* en *n* subconjuntos, {S1, S2,... Sn}, donde *Sj* contiene esos ejemplos de *T* que tienen el valor *aj* en *A* y sij el número de ejemplos de la clase Ci en el subconjunto Sj. E(A) se calcula:

$$E(A)= \sum_{j=1}^{n} sij/ s * I(Sij)$$

y sij el número de ejemplos de la clase Ci en el subconjunto Sj

$$I(Sij)= -\sum_{j=1}^{m} pij \log2(pij)$$

donde, pij=sij / |sj| es la probabilidad que un ejemplo de Sj pertenezca a la clase Ci.

En otras palabras, Gain(T, A), es la reducción esperada de la entropía causada por el particionamiento de T de acuerdo con el atributo A. Finalmente las reglas de clasificación se obtienen recorriendo cada rama del árbol desde la raíz hasta el nodo terminal. El antecedente de la regla es la conjunción de los pares recogidos en cada nodo y el consecuente es el nodo terminal.

**4.2.3 Modelo Relacional.** Un *dominio* es un conjunto de valores del mismo tipo. Un dominio es *simple* si todos sus valores son atómicos (i.e. no se pueden descomponer).

El producto cartesiano de una serie de dominios *D1,D2, . . . , Dn* , n (n>0) (no necesariamente diferentes) denotado *D1 x D2 x . . . x Dn* es el conjunto de todas las n-tuplas *<v1, v2, . . . , vn >* tales que, para cualquier i, i = 1, . . ., n se cumple que vi ∈ Di. Una relación *R* es un subconjunto del producto cartesiano de estos *n* dominios:

$$R \subseteq D1 \times D2 \times ...xDn$$

La *Cardinalidad* de una relación R es el número *m* de tuplas que contiene R y puede variar con el tiempo.

El grado de una relación R es el número de dominios *n* (n>0) que intervienen en la definición de R y el cual permanece constante.

Una relación R está compuesta por su esquema y su extensión. Un esquema de una relación *R* denotada como *R(A)* es un conjunto de n atributos *A* definidos sobre el conjunto de dominios *D*:

$$R(A)= R\{A1 : D1, A2 : D2,...,An : Dn\}$$

Una extensión o instancia de relación (llamada comúnmente tabla) *r(A)*, definida sobre el esquema *R(A)* es un conjunto de *m* tuplas {t1,t2,...,tm}, donde cada tupla *t(A)* es un conjunto de *n* pares atributo-valor (A1:v1, A2:v2, . . . , Ai: vi, . . ., An:vn} donde *Vi* es el valor i del dominio *Di* asociado al atributo *Ai.*

La tabla *r(A)*, definida sobre el esquema *R(A)*, de grado *n* y cardinalidad *m* esta constituida por un conjunto de m tuplas, n-arias y con A atributos.

$$r(A) = \{ ti(A) \mid ti = <vi(A1),vi(A2),.....vi(An)> , \ vi(Aj) \in Dj, i=1..\ m, \ j=1\ ..\ n\}$$

Dos o más relaciones son unión compatibles o del mismo tipo si tienen atributos equivalentes en número y dominios.

Una función agregada *f*, sobre un atributo Aj de una relación R, *f*(Aj)(R), Aj ∈ R(A), retorna un valor, que se obtiene de aplicar la función *f* al conjunto de valores *v(Aj)* del atributo Aj en r(A).

$$f(Aj)(R)=\{y \mid y= f(Vi(Aj)) \ \ i=1\ ..m\}$$

Un operador relacional de agregación *Agg* sobre un atributo Aj de una relación R, Agg(Aj)(R), Aj ∈ R(A), es un operador que aplica una función agregada *f* sobre el atributo Aj y producen una nueva relación cuyo esquema está formado por un atributo y cuya extensión esta formada por una sola tupla. Esto con el fin de conservar la propiedad de cierre del álgebra relacional.

$$Agg(Aj)(R)=\{t \mid t = f(Aj)(R)\}$$

## 4.3 OPERADORES UNARIOS DEL ÁLGEBRA RELACIONAL PARA ASOCIACIÓN

### 4.3.1 Operador Associator ( a ). Sintaxis:

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

El operador *Associator* genera, por cada tupla de la relación *R*, todos sus posibles subconjuntos (itemsets) de diferente tamaño. *Associator* toma cada tupla *t* de *R* y dos parámetros :<tamaño_inicial> y <tamaño_final> como entrada, y retorna, por cada tupla *t*, las diferentes combinaciones de atributos Xi, de tamaño <tamaño_inicial> hasta tamaño <tamaño_final>, como tuplas en una nueva relación. El orden de los atributos en el esquema de *R* determina los atributos en los subconjuntos con valores, el resto se hacen nulos. El tamaño máximo de un itemset y por consiguiente el tamaño final máximo (<tamaño_final>) que se puede tomar como entrada es el correspondiente al valor del grado de la relación.

Formalmente, sea $A=\{A_1,..., A_n\}$ el conjunto de atributos de la relación R de grado n y cardinalidad m, IS y ES el tamaño inicial y final respectivamente de los subconjuntos a obtener. El operador α aplicado a R

$$\alpha_{IS,\,ES}(R) = \{ \cup_{all} X_i \mid X_i \subseteq t_i,\ t_i \hat{I} R,\ \forall_i \forall_k\ (X_i = <v_i(A_1), v_i(A_2), null, \ldots, v_i(A_k), null>,\ v_i(A_k) <> null),\ (i = (2^n - 1) * m),\ (k = IS \ldots ES),\ A_1 < A_2 < \ldots < A_k,\ IS = 1,\ ES = n \}$$

produce una nueva relación cuyo esquema R(A) es el mismo de R de grado *n* y cardinalidad *m'= (2^n -1) * m* y cuya extensión r(A) está formada por todos los subconjuntos Xi generados a partir de todas las combinaciones posibles de los valores no nulos $v_i(A_k)$ de los atributos de cada tupla $t_i$ de R. En cada tupla Xi únicamente un grupo de atributos mayor o igual que IS y menor o igual que ES tienen valores, los demás atributos se hacen nulos.

Ejemplo 4.1. Sea la relación R(A,B,C,D) :

**Tabla 4.1. Relación R con los atributos A, B, C, D, ejemplo operador Associator**

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

El resultado de $R1=\alpha_{2,4}(R)$ es el siguiente:

**Tabla 4.2. Resultado R1 del operador Associator sobre R**

| 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 |

**4.3.2 Operador EquiKeep (χ ).** Sintaxis:

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

El operador *EquiKeep* restringe los valores de los atributos de cada una de las tuplas de la relación *R* a únicamente los valores de los atributos que satisfacen una expresión lógica <*expresión_lógica*>, la cual esta formada por un conjunto de cláusulas de la forma *Atributo=Valor*, y operaciones lógicas AND, OR y NOT. En cada tupla, los valores de los atributos que no cumplen la condición <*expresión_lógica*> se hacen nulos. *EquiKeep* elimina las tuplas vacías, i.e. las tuplas con todos los valores de sus atributos nulos.

Formalmente, sea $A=\{A_1,\ldots, A_n\}$ el conjunto de atributos de la relación *R* de esquema R(A), de grado *n* y cardinalidad *m*. Sea *p* una expresión lógica integrada por cláusulas de la forma $A_i = const$, unidas por los operadores booleanos AND (^), OR (v), NOT (¬). El operador χ aplicado a la relación *R* con la expresión lógica *p*

$$\chi_p(R)=\{ t_i(A) \mid \forall i \forall j\ (p(v_i(A_j))= v_i(A_j) \text{ si p =true y } p(v_i(A_j))= \text{null si p =false}),\ i=1..m',\ j=1..n,\ m' \le m\}$$

produce una relación de igual esquema R(A) de grado *n* y cardinalidad *m'*, donde $m' \le m$. En su extensión, cada n-tupla *ti*, esta formada por los valores de los atributos de R, $v_i(A_j)$, que cumplan la expresión lógica p, es decir $p(v_i(Y_j))$ es verdadero, y por valores nulos si $p(v_i(Y_j))$ es falso.

Ejemplo 4.2. Sea la relación R(A,B,C,D) y la operación $R1=\chi_{A=a1 \text{ v } B=b1 \text{ v } C=c2 \text{ v } D=d1}(R)$

**Tabla 4.3. Relación R con los atributos A, B, C, D, ejemplo operador EquiKeep**

| 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 |

El resultado de esta operación es la siguiente:

**Tabla 4.4. Resultado R1 del operador EquiKeep sobre 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 |

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

## 4.4 OPERADORES DEL ÁLGEBRA RELACIONAL PARA CLASIFICACIÓN

### 4.4.1 Operador Mate ( $\mu$ ).

Sintaxis:

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

El operador *Mate* genera, por cada una de las tuplas de la relación *R*, todas los posibles combinaciones formadas por los valores no nulos de los atributos pertenecientes a una lista de atributos denominados Atributos Condición, *<lista_atributos_condición>* y el valor no nulo del atributo denominado Atributo Clase, *<atributo_clase>*. *Mate* toma como entrada cada tupla de *R* y produce una nueva relación cuyo esquema esta formado por los *Atributos Condición* y el *atributo Clase*, con tuplas formadas por todas las posibles combinaciones de los atributos de la lista de atributos LC con el atributo clase Ac, los demás valores de los atributos se hacen nulos.

Formalmente, sea $A = \{A_1, \ldots, A_n\}$ el conjunto de atributos de la relación *R* de grado *n* y cardinalidad *m*, $LC \subset A$, $LC \neq \phi$ la lista de atributos condición a emparejar y *n'* el número de atributos de LC, $|LC| = n'$, $n' < n$, $Ac \in A$ , $Ac \cap L = \phi$ el atributo clase con el que se emparejarán los atributos de LC. El operador $\mu$ aplicado a la lista de atributos LC, al atributo clase Ac de la relación R:

$$\mu_{LC;\,Ac}(R) = \{ ti(M) \mid M=LC \cup Ac,\ LC \subset A,\ |LC| = n',\ n'<n,\ Ac \in A,\ Ac \cap LC = \phi,\ ti=Xi,\ ),\ i= 1..m',\ m'= (2^{n'}-1)*m,\ \forall_i \forall_k (\ X_i=<null, ...,v_i(A_k)... ,null,...,v_i(Ac)>,\ v_i(A_k)\ v_i(Ac) <>null),\ k=1..\ n'\ \}$$

produce una relación cuyo esquema es R(M), $M=LC \cup Ac$, de grado *g*, $g= n'+1$ y cuya extensión r(M) de cardinalidad m', $m'= (2^{n'}-1)*m$ es el conjunto de g-tuplas *ti*, tal que en cada g-tupla únicamente los atributos que forman la combinación $X_i \in LC$, $(k =1..n')$ y el atributo Ac tienen valor, el resto de atributos se hacen nulos.

Ejemplo 4.3. Sea la relación R(A,B,C,D):

**Tabla 4.5. Relación R con los atributos A, B, C, D, ejemplo operador Mate**

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

El resultado de la operación $R1=\mu_{A,B,C;D}(R)$ es:

**Tabla 4.6. Resultado R1 del operador Mate sobre R.**

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

**4.4.2 Función agregada Entro().** Sintaxis :

*Entro*(atributo; atributo_clase;R)

La función *Entro()* permite calcular la entropía de una relación R con respecto a un atributo <atributo> y un atributo clase <atributo_clase>.

Formalmente, sea $A=\{A_1, ..., A_{n,Ac}\}$ el conjunto de atributos de la relación *R* con esquema *R(A)*, extensión *r(A)*, grado *n* y cardinalidad *m*. Sea *t* el número de distintos valores del <atributo_clase> $Ac \hat{I} R(A)$ que divide a *r(A)* en *t* diferentes clases, $Ci\ (i=1...t)$. Sea *ri* el número de tuplas de *r(A)* que pertenecen a la clase *Ci*. Sea *q* el número de distintos valores del <atributo> $Ak \hat{I} R(A), \{v1(Ak),v2(Ak),..,vq(Ak)\}$ el cual particiona a *r(A)* en *q* subconjuntos, *Sj* (j=1...q). *Sk* contiene todos las tuplas de r(A) que tienen el valor vj(Ak) del atributo Ak. Sea sij el número de tuplas de la clase Ci en el subconjunto Sj. La función *Entro*(Ak; Ac; R), retorna la entropía de R con respecto al atributo Ak, que se obtiene de la siguiente manera :

$$Entro(Ak; Ac; R)=\{y \mid y= -\sum_{i=1..t, j=1..q} pij \log2(pij),\ pij=sij/|Sj|\}$$

donde pij=sij/|Sj| es la probabilidad que una tupla en Sj pertenezca a la clase Ci.

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

$$Entro(Ac;Ac; R)=\{y \mid y= -\sum_{i=1..t} pi \log2(pi),\ pi= ri/m\}$$

donde:
- pi es la probabilidad que una tupla cualquiera pertenezca a la clase Ci
- *ri* el número de tuplas de *r(A)* que pertenecen a la clase *Ci.*

**4.4.3 Función agregada Gain().** Sintaxis:

*Gain*(atributo;atrib_clase;R)

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

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

donde:

*Entro* (Ac; Ac; R) es la entropía de la relación R con respecto al atributo clase Ac.
*Entro*(Ak;Ac; R) es la entropía de la relación R con respecto al atributo Ak .

## 4.5 PRIMITIVAS SQL PARA ASOCIACION

**4.5.1 Primitiva Associator Range.** Esta primitiva implementa el operador algebraico Associator en la cláusula SQL SELECT. Associator 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, *Associator* tiene la siguiente sintaxis:

SELECT <ListaAtributosTablaDatos> [INTO <NombreTablaAssociator>]
FROM <NombreTablaDatos>
WHERE <CláusulaWhere>
ASSOCIATOR RANGE <numero1> UNTIL <numero2>
GROUP BY <ListaAtributosTablaDatos>

<ListaAtributosTablaDatos>:=<Atributo> | <ListaAtributos>

<numero1>:= 1, 2, 3, 4...
<numero2>:= 1, 2, 3, 4...

donde:

La cláusula SELECT <ListaAtributosTablaDatos> permite seleccionar los atributos de la tabla de datos <NombreTablaDatos> cuyos valores formarán los diferentes subconjuntos o itemsets como tuplas de la nueva tabla <NombreTablaAssociator>.

La cláusula INTO <NombreTablaAssociator> define el nombre de la tabla <NombreTablaAssociator> donde se almacenarán los resultados de ASSOCIATOR con los atributos especificados en la cláusula SELECT <ListaAtributosTablaDatos> como esquema. Si no se especifica la cláusula INTO, el resultado se almacenará en una tabla temporal como sucede normalmente en el lenguaje SQL.

La cláusula FROM <NombreTablaDatos> WHERE <CláusulaWhere> determina el nombre de la tabla <NombreTablaDatos> de donde se extraerá el conjunto de tuplas que cumplan las restricciones de la <CláusulaWhere> para formar las diferentes combinaciones o itemsets.

La cláusula ASSOCIATOR RANGE <número1> UNTIL <número2> determina el tamaño inicial hasta el final de los diferentes subconjuntos o itemsets que formará el operador ASSOCIATOR.

La cláusula GROUP BY <ListaAtributosTablaDatos> agrupa la tabla <NombreTablaDatos> por los atributos especificados en <ListaAtributosTablaDatos>.

La primitiva ASSOCIATOR facilita el cálculo de los itemsets frecuentes para el descubrimiento de Reglas de Asociación en tablas multicolumna como se muestra en el siguiente ejemplo:

Ejemplo 4.4. Sea la tabla ESTUDIANTES (PROGRAMA, EDAD, SEXO, ESTRATO, PROMEDIO) que se encuentra discretizada.

**Tabla 4.7. Tabla discretizada ESTUDIANTES.**

| PROGRAMA | EDAD | SEXO | ESTRATO | PROMEDIO |
|---|---|---|---|---|
| Sistemas | 16..20 | M | 2 | 3.8 |
| Idiomas | 21..25 | F | 3 | 3.9 |
| Sistemas | 16..20 | F | 3 | 3.9 |
| Física | 21..25 | M | 2 | 3.5 |
| Psicología | 21..25 | F | 4 | 4.0 |

Calcular el soporte para las itemsets de tamaño 2 y 3 formados por los atributos PROGRAMA, SEXO, ESTRATO y almacenar los resultados en la tabla assostudent.

Esta consulta se realiza 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

Para implementar la primitiva ASSOCIATOR RANGE en el interior de un SGBD y optimizarla el proceso que se sigue es el siguiente:

Inicialmente se ejecuta la cláusula SELECT. En este caso se seleccionan los atributos *programa, sexo y estrato.* Luego se ejecuta la cláusula ASSOCIATOR RANGE, es decir, se procede a generar los itemsets de tamaño 2 y 3 como se estipula en la primitiva ASSOCIATOR RANGE, así:

**Tabla 4.8. Itemsets de tamaño 2 y 3 generados con la cláusula Associator Range sobre los atributos PROGRAMA, SEXO, ESTRATO**

| PROGRAMA | SEXO | ESTRATO |
|---|---|---|
| Sistemas | M | null |
| Sistemas | null | 2 |
| null | M | 2 |
| Sistemas | M | 2 |
| Idiomas | F | null |
| Idiomas | null | 3 |
| null | F | 3 |
| Idiomas | F | 3 |
| Sistemas | F | null |
| Sistemas | null | 3 |
| null | F | 3 |
| Sistemas | F | 3 |
| Física | M | null |
| Física | null | 2 |
| null | M | 2 |
| Física | M | 2 |
| Psicología | F | null |
| Psicología | null | 4 |
| null | F | 4 |
| Psicología | F | 4 |

Luego se agrupa por *programa, sexo y estrato* de acuerdo a la cláusula GROUP BY y se calcula la función agregada COUNT(). El resultado se almacena en la tabla ASOSTUDENT:

**Tabla 4.9. Tabla ASOSTUDENT**

| PROGRAMA | SEXO | ESTRATO | SOPORTE |
|---|---|---|---|
| Física | M | 2 | 1 |
| Física | M | null | 1 |
| Física | null | 2 | 1 |
| Idiomas | F | 3 | 1 |
| Idiomas | F | null | 1 |
| Idiomas | null | 3 | 1 |
| Psicología | F | 4 | 1 |
| Psicología | F | null | 1 |
| Psicología | null | 4 | 1 |
| Sistemas | F | 3 | 1 |
| Sistemas | F | null | 1 |
| Sistemas | M | 2 | 1 |
| Sistemas | M | null | 1 |
| Sistemas | null | 2 | 1 |
| Sistemas | null | 3 | 1 |
| null | F | 3 | 2 |
| null | F | 4 | 1 |
| null | M | 2 | 2 |

Si se desea obtener los itemsets frecuentes de tamaño 2 y 3 que cumplan con un soporte mayor o igual a 2, el resultado sería el siguiente:

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

**Tabla 4.10. Tabla ASOSTUDENT con soporte mayor o igual a 2**

| PROGRAMA | SEXO | ESTRATO | SOPORTE |
|---|---|---|---|
| null | F | 3 | 2 |
| null | M | 2 | 2 |

**4.5.2 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 sintaxis:

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

<ListaAtributosTablaDatos>:=<Atributo> | <ListaAtributos>
<CondiciónValoresAtributos>:=<Atributo=Valor><operador><Atributo=Valor>
<operador>:=AND,OR,NOT,IN

donde:

La cláusula SELECT <ListaAtributosTablaDatos> permite seleccionar los atributos de la tabla de datos <NombreTablaDatos> cuyos valores se conservarán si cumplen la condición <CondiciónValoresAtributos>, especificada en la primitiva EquiKeep**.**

La cláusula INTO <NombreTablaEquiKeep> define el nombre de la tabla <NombreTablaEquiKeep> donde se almacenarán los resultados de EQUIKEEP con los atributos <ListaAtributosTablaDatos> como esquema. Si no se especifica la cláusula INTO, el resultado se almacenará en una tabla temporal.

La cláusula FROM <NombreTablaDatos> WHERE <CláusulaWhere> determina el nombre de la tabla de datos <NombreTablaDatos> con el conjunto de tuplas que cumplan las restricciones de la <CláusulaWhere>.

La primitiva EQUIKEEP ON <CondiciónValoresAtributos > conserva los valores de los atributos de <ListaAtributosTablaDatos> que cumplan la condición especificada en <CondiciónValoresAtributos >. El resto de valores de los atributos se hacen nulos.

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 en el algoritmo Apriori, en el FP-Tree o conjuntamente con cualquiera de las primitivas de Asociación (Associator, Assorow, Assocol).

Ejemplo 4.5. Sea la tabla TRANSACCION (TID, ID_ITEM1, ID_ITEM2, ID_ITEM3)

**Tabla 4.11. Tabla TRANSACCION**

| TID | ID_ITEM1 | ID_ITEM2 | ID_ITEM3 |
|---|---|---|---|
| 100 | i1 | i2 | |
| 200 | i2 | i3 | i4 |
| 300 | i1 | i2 | i3 |
| 400 | i2 | i3 | i4 |
| 500 | i3 | i4 | |

Suponiendo que ya se han calculado los itemsets frecuentes de tamaño 1 con un soporte = 3, los cuales son {i2}, {i3}, {i4}. Calcular el soporte para las itemsets de tamaño 2 y 3 formados por lo atributos ID_ITEM1, ID_ITEM2, ID_ITEM3 y almacenar los resultados en la tabla tran23.

La sentencia SQL que permite obtener esta consulta es la siguiente:

```
SELECT id_item1, id_item2,  id_item3, count(*) AS soporte  INTO tran23
FROM transaccion
EQUIKEEP ON id_item1 IN (i2,i3,i4) OR id_item2 IN (i2,i3,i4) OR id_item3 IN (i2,i3,i4)
ASSOCIATOR RANGE  2 UNTIL 3
GROUP BY id_item1, id_item2,  id_item3
```

Para implementar la primitiva EQUIKEEP ON al interior de un SGBD y optimizarla, se debe tener en cuenta que inicialmente se ejecuta la cláusula SELECT, en este caso se seleccionan los atributos *id_item1*, *id_item2*, *id_item3* de la tabla transacción. Luego sobre la tabla resultante, se ejecuta la primitiva EQUIKEEP ON que conserva los valores de los atributos que cumplan la condición *id_item1 IN (i2,i3,i4) OR id_item2 IN (i2,i3,i4) OR id_item3 IN (i2,i3,i4)* así:

**Tabla 4.12. Primitiva Equikeep sobre los atributos ID_ITEM1, ID_ITEM2, ID_ITEM3**

| ID_ITEM1 | ID_ITEM2 | ID_ITEM3 |
|---|---|---|
| null | i2 | null |
| i2 | i3 | i4 |
| null | i2 | i3 |
| i2 | i3 | i4 |
| i3 | i4 | null |

Posteriormente se generan los itemsets de tamaño 2 y 3 con la primitiva ASSOCIATOR RANGE 2 UNTIL 3, así:

**Tabla 4.13. Itemsets de tamaño 2 y 3 generados con la primitiva Associator Range sobre los atributos ID_ITEM1, ID_ITEM2, ID_ITEM3**

| ID_ITEM1 | ID_ITEM2 | ID_ITEM3 |
|---|---|---|
| i2 | i3 | null |
| i2 | null | i4 |
| null | i3 | i4 |
| i2 | i3 | i4 |
| null | i2 | i3 |
| i2 | i3 | null |
| i2 | null | i4 |
| null | i3 | i4 |
| i2 | i3 | i4 |
| i3 | i4 | null |

y finalmente se ejecuta cláusula GROUP BY, se cuenta la frecuencia de los itemsets y se genera la tabla resultante TRAN23:

**Tabla 4.14. Tabla TRAN23**

| ID_ITEM1 | ID_ITEM2 | ID_ITEM3 | SOPORTE |
|---|---|---|---|
| i2 | i3 | null | 2 |
| i2 | null | i4 | 2 |
| null | i3 | i4 | 2 |
| i2 | i3 | i4 | 2 |
| null | i2 | i3 | 1 |
| i3 | i4 | null | 1 |

Si se desea obtener las itemsets frecuentes de tamaño 2 y 3, con soporte mayor o igual a 1, la sentencia sería:

```
SELECT  id_item1, id_item2,  id_item3, count(*) AS soporte  INTO tran23
FROM transaccion
EQUIKEEP ON id_item1 IN (i2,i3,i4) OR id_item2 IN (i2,i3,i4) OR id_item3 IN (i2,i3,i4)
ASSOCIATOR RANGE  2 UNTIL  3
GROUP BY id_item1, id_item2, id_item3 HAVING count(*) >= 1
```

La tabla TRAN23 resultante sería:

**Tabla 4.15. Tabla TRAN23 con soporte mayor o igual a 1**

| ID_ITEM1 | ID_ITEM2 | ID_ITEM3 | SOPORTE |
|---|---|---|---|
| i2 | i3 | null | 2 |
| i2 | null | i4 | 2 |
| Null | i3 | i4 | 2 |
| i2 | i3 | i4 | 2 |

## 4.9 PRIMITIVAS SQL PARA CLASIFICACION

**4.6.1 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 en cada registro forma todas las posibles combinaciones con otro atributo de la misma tabla denominado atributo clase.

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>

<ListaAtributosTablaDatos>:=<ListaAtributos><Atributo>
<ListaAtributos>:=<Atributo> | <ListaAtributos>
<ListaAtributosCondicion>:=<ListaAtributos><Atributo>
<AtributoClase>:=<Atributo>

donde:

La cláusula SELECT <ListaAtributosTablaDatos> permite seleccionar de la tabla <NombreTablaDatos> tanto el conjunto de atributos que serán combinados <ListaAtributosCondicion> como el atributo con el cual se combinarán <AtributoClase>.

La cláusula INTO <NombreTablaMate> define el nombre de la tabla <NombreTablaMate> donde se almacenarán los resultados de la primitiva MATE cuyo esquema estará determinado por los atributos de <ListaAtributosTablaDatos>. Si no se especifica la cláusula INTO, el resultado se almacenará en una tabla temporal.

La cláusula FROM <NombreTablaDatos> WHERE <CláusulaWhere> determina el nombre de la tabla de datos de entrada <NombreTablaDatos> con las restricciones <CláusulaWhere> que las tuplas deben cumplir.

La cláusula MATE BY <ListaAtributosCondicion> WITH <AtributoClase> determina el conjunto de atributos <ListaAtributosCondicion> con los cuales el atributo clase <AtributoClase> se combinará.

La cláusula GROUP BY <ListaAtributosTablaDatos> agrupa la tabla de entrada <NombreTablaDatos> por los atributos <ListaAtributosTablaDatos>.

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 4.6. Sea la tabla SINTOMAS (SID, D_CABEZA, D_MUSCULAR, TEMPERATURA, GRIPA).

**Tabla 4.16. Tabla SINTOMAS**

| SID | D_CABEZA | D_MUSCULAR | TEMPERATURA | GRIPA |
|---|---|---|---|---|
| 1 | No | Si | Alta | Si |
| 2 | Si | No | Alta | Si |
| 3 | Si | Si | Media | No |
| 4 | No | Si | Normal | Si |
| 5 | Si | No | Media | No |
| 6 | No | No | Normal | No |
| 7 | Si | No | Normal | Si |
| 8 | Si | Si | Alta | Si |

Obtener en una sola consulta y con una sola búsqueda sobre la tabla sintomas, las ocurrencias de las diferentes combinaciones de los atributos d_cabeza, 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_cabeza, d_muscular, temperatura, gripa, count(*) INTO clasesintomas
FROM sintomas
MATE BY d_cabeza, d_muscular, temperatura WITH gripa
GROUP BY d_cabeza, d_muscular, temperatura, gripa
```

Para ejecutar esta orden, el optimizador toma cada registro de la tabla y forma todas las posibles combinaciones de los atributos condición con el atributo clase usando la primitiva MATE BY d_cabeza, d_muscular, temperatura WITH gripa, así:

**Tabla 4.17. Ejecución Primitiva Mate By con el atributo clase gripa.**

| D_CABEZA | D_MUSCULAR | TEMPERATURA | GRIPA |
|---|---|---|---|
| No | null | null | Si |
| null | Si | null | Si |
| null | null | Alta | Si |
| No | Si | null | Si |
| No | null | Alta | Si |
| null | Si | Alta | Si |
| No | Si | Alta | Si |
| Si | null | null | Si |
| null | No | null | Si |
| null | null | Alta | Si |
| Si | No | null | Si |
| Si | null | Alta | Si |
| null | No | Alta | Si |
| Si | No | Alta | Si |
| Si | null | null | No |
| null | Si | null | No |
| null | null | Media | No |
| Si | Si | null | No |
| Si | null | Media | No |
| null | Si | Media | No |
| Si | Si | Media | No |
| No | null | null | Si |
| null | Si | null | Si |
| null | null | Normal | Si |
| No | Si | null | Si |
| No | null | Normal | Si |
| null | Si | Normal | Si |
| No | Si | Normal | Si |
| Si | null | null | No |
| null | No | null | No |
| null | null | Media | No |
| Si | No | null | No |
| Si | null | Media | No |
| null | No | Media | No |
| Si | No | Media | No |
| No | null | null | No |
| null | No | null | No |
| null | null | Normal | No |
| No | No | null | No |
| No | null | Normal | No |
| null | No | Norma1 | No |

| | | | |
|---|---|---|---|
| No | No | Normal | No |
| Si | null | null | No |
| null | No | null | No |
| null | null | Normal | No |
| Si | No | null | No |
| Si | null | Normal | No |
| null | No | Normal | No |
| Si | No | Normal | No |
| Si | null | null | Si |
| null | Si | null | Si |
| null | null | Alta | Si |
| Si | Si | null | Si |
| Si | null | Alta | Si |
| null | Si | Alta | Si |
| Si | Si | Alta | Si |

A la tabla resultante, la agrupa con la cláusula GROUP BY, cuenta la frecuencia de las parejas, proyecta los atributos seleccionados en la cláusula SELECT y finalmente genera la tabla CLASESINTOMAS:

**Tabla 4.18. Tabla CLASESINTOMAS**

| D_CABEZA | D_MUSCULAR | TEMPERATURA | GRIPA | COUNT |
|---|---|---|---|---|
| No | No | Normal | No | 1 |
| No | No | null | No | 1 |
| No | Si | Alta | Si | 1 |
| No | Si | Normal | Si | 1 |
| No | Si | null | Si | 2 |
| No | null | Alta | Si | 1 |
| No | null | Normal | No | 1 |
| No | null | Normal | Si | 1 |
| No | null | null | No | 1 |
| No | null | null | Si | 2 |
| Si | No | Alta | Si | 1 |
| Si | No | Media | No | 1 |
| Si | No | Normal | No | 1 |
| Si | No | null | No | 2 |
| Si | No | null | Si | 1 |
| Si | Si | Alta | Si | 1 |
| Si | Si | Media | No | 1 |
| Si | Si | null | No | 1 |
| Si | Si | null | Si | 1 |

| | | | | |
|---|---|---|---|---|
| Si | null | Alta | Si | 2 |
| Si | null | Media | No | 2 |
| Si | null | Normal | No | 1 |
| Si | null | null | No | 3 |
| Si | null | null | Si | 2 |
| null | No | Alta | Si | 1 |
| null | No | Media | No | 1 |
| null | No | Normal | No | 2 |
| null | No | null | No | 3 |
| null | No | null | Si | 1 |
| null | Si | Alta | Si | 2 |
| null | Si | Media | No | 1 |
| null | Si | Normal | Si | 1 |
| null | Si | null | No | 1 |
| null | Si | null | Si | 3 |
| null | null | Alta | Si | 3 |
| null | null | Media | No | 2 |
| null | null | Normal | No | 2 |
| null | null | Normal | Si | 1 |

**4.6.2 Función SQL agregada Entro().** Esta función SQL implementa la función agregada Entro() en la cláusula SQL SELECT. *Entro()* permite calcular, conjuntamente con la primitiva Mate by, la entropía de una tabla con respecto a un atributo condición y un atributo clase.

Ejemplo 4.7. Calcular la entropía para los atributos condición *d_cabeza, d_muscular, temperatura* de la tabla SINTOMAS del ejemplo anterior.

La orden SQL que permite calcular la entropía para cada atributo condición es la siguiente:

```
SELECT DISTINCT d_cabeza, d_muscular, temperatura, entro(*) INTO entrosintomas
FROM sintomas
MATE BY d_cabeza, d_muscular, temperatura WITH gripa
GROUP BY d_cabeza, d_muscular, temperatura, gripa
```

Para ejecutar esta orden, el optimizador empareja cada atributo condición con el atributo clase usando la primitiva MATE BY d_cabeza, d_muscular, temperatura WITH gripa. A la tabla resultante, la agrupa con la cláusula GROUP BY, calcula la entropía y proyecta los atributos seleccionados en la cláusula SELECT, eliminando los repetidos y finalmente genera la tabla ENTROSINTOMAS.

**Tabla 4.19. Tabla ENTROSINTOMAS**

| D_CABEZA | D_MUSCULAR | TEMPERATURA | ENTRO() |
|---|---|---|---|
| No | No | Normal | $0+1/1\log_2(1/1)$ |
| No | No | null | $0+1/1\log_2(1/1)$ |
| No | Si | Alta | $1/1\log_2(1/1)+0$ |
| No | Si | Normal | $1/1\log_2(1/1)+0$ |
| No | Si | null | $2/2\log_2(2/2)+0$ |
| No | null | Alta | $1/1\log_2(1/1)+0$ |
| No | null | Normal | $1/2\log_2(1/2)+1/2\log_2(1/2)$ |
| No | null | null | $2/3\log_2(2/3)+1/3\log_2(1/3)$ |
| Si | No | Alta | $1/1\log_2(1/1)+0$ |
| Si | No | Media | $0+1/1\log_2(1/1)$ |
| Si | No | Normal | $0+1/1\log_2(1/1)$ |
| Si | No | null | $1/3\log_2(1/3)+2/3\log_2(2/3)$ |
| Si | Si | Alta | $1/1\log_2(1/1)+0$ |
| Si | Si | Media | $0+1/1\log_2(1/1)$ |
| Si | Si | null | $1/2\log_2(1/2)+1/2\log_2(1/2)$ |
| Si | null | Alta | $2/2\log_2(2/2)+0$ |
| Si | null | Media | $0+2/2\log_2(2/2)$ |
| Si | null | Normal | $0+1/1\log_2(1/1)$ |
| Si | null | null | $2/5\log_2(2/5)+3/5\log_2(3/5)$ |
| null | No | Alta | $1/1\log_2(1/1)+0$ |
| null | No | Media | $0+1/1\log_2(1/1)$ |
| null | No | Normal | $0+2/2\log_2(2/2)$ |
| null | No | null | $1/4\log_2(1/4)+3/4\log_2(3/4)$ |
| null | Si | Alta | $2/2\log_2(2/2)+0$ |
| null | Si | Media | $0+1/1\log_2(1/1)$ |
| null | Si | Normal | $1/1\log_2(1/1)+0$ |
| null | Si | null | $3/4\log_2(3/4)+1/4\log_2(1/4)$ |
| null | null | Alta | $3/3\log_2(3/3)+0$ |
| null | null | Media | $0+2/2\log_2(2/2)$ |
| null | null | Normal | $1/3\log_2(1/3)+2/3\log_2(2/3)$ |

**4.6.3 Función SQL agregada Gain().** Esta función SQL implementa la función agregada Gain() en la cláusula SQL SELECT. *Gain()* permite calcular, conjuntamente con la primitiva Mate by, la ganancia de información de una tabla con respecto a un atributo condición y un atributo clase.

Ejemplo 4.8. Calcular la ganancia de información para los atributos condición *d_cabeza, d_muscular, temperatura* de la tabla SINTOMAS del ejemplo 4.7.

La orden SQL que permite calcular la ganancia de información para cada atributo condición es la siguiente:

SELECT gain(d_cabeza) AS Gd_cabeza, gain(d_muscular) AS Gd_muscular, gain(temperatura) AS G_temperatura INTO gainsintomas
FROM sintomas
MATE BY d_cabeza, d_muscular, temperatura WITH gripa

Para ejecutar esta orden, el optimizador empareja cada atributo condición con el atributo clase usando la primitiva MATE BY d_cabeza, d_muscular, temperatura WITH gripa. Para el cálculo de la ganancia de información de cada atributo, internamente el optimizador calcula la entropía y con este resultado, calcula la ganancia de información, las proyecta por la cláusula SELECT, y finalmente genera la tabla GAINSINTOMAS.

**Tabla 4.20. Tabla GAINSINTOMAS**

| GD_CABEZA | GD_MUSCULAR | G_TEMPERATURA |
|---|---|---|
| null | null | Alta |
| null | Si | null |

## 4.7 OPERADORES SQL PARA ASOCIACIÓN Y CLASIFICACIÓN

**4.7.1 Operador Describe Association Rules.** Este operador genera todas las reglas de asociación que cumplen con un soporte y una confianza mayor o igual que unos determinados umbrales especificados por el usuario. *Describe* puede generar reglas de asociación unidimensionales o multidimensionales una vez se hayan hallado los itemsets frecuentes.

**4.7.2 Reglas de Asociación Unidimensionales.** La sintaxis para generar reglas de asociación unidimensionales utilizando una tabla de datos multicolumna con todos sus atributos del mismo tipo es la siguiente:

DESCRIBE UNIDIMENSIONAL ASSOCIATION RULES
FROM <NombreTablaItemsetsFrecuentes> [INTO <TablaReglasAsociación>]
WITH SUPPORT <numero1> AND CONFIDENCE <numero2>
[AS <SubconsultaCalculoItemsetsFrecuentes>]

<numero1>:=1,2,3,4,….
<numero2>:=1,2,3,4,….
<subconsulta>:=<SELECT FROM WHERE EQUIKEEP ASSOCIATOR GROUP BY>

donde:

La cláusula DESCRIBE UNIDIMENSIONAL ASSOCIATION RULES FROM <NombreTablaItemsetsFrecuentes> especifica el nombre de la tabla de datos <NombreTablaItemsetsFrecuentes> donde se han calculado los itemsets frecuentes.

La cláusula opcional INTO <TablaReglasAsociación> permite almacenar en una tabla <TablaReglasAsociación> las reglas de asociación que se generen, para posteriores consultas.

La cláusula WITH SUPPORT <numero1> AND CONFIDENCE <numero2> especifica los valores del soporte mínimo <numero1> y la confianza mínima <numero2> utilizados para el cálculo de los itemsets frecuentes y la generación de las reglas de asociación respectivamente.

La cláusula opcional [AS <SubconsultaCalculoItemsetsFrecuentes>] permite especificar conjuntamente con la cláusula DESCRIBE, la subconsulta <SubconsultaCalculoItemsetsFrecuentes> que halla los itemsets frecuentes. Para efectos de optimización del operador DECRIBE ASSOCIATION RULES, la subconsulta se ejecutaría primero y luego se generarían las reglas con DECRIBE ASSOCIATION RULES. En el caso de que esta cláusula no se especifique, los procesos de cálculo de los itemsets frecuentes y el de generación de reglas se realizarían de manera independiente. Primero se encontrarían los itemsets frecuentes mediante una sentencia <SELECT FROM WHERE EQUIKEEP ASSOCIATOR GROUP BY HAVING > y posteriormente se generarían las reglas de asociación con DESCRIBE.

La subconsulta <SubconsultaCalculoItemsetsFrecuentes> tendría la siguiente sintaxis:

```
SELECT <ListaAtributosTablaDatos> [INTO <NombreTablaAssociator>]
FROM <NombreTablaDatos> WHERE <CláusulaWhere>
EQUIKEEP ON <CondiciónValoresAtributos >
ASSOCIATOR RANGE <numero1> UNTIL <numero2>
GROUP BY <ListaAtributosTablaDatos>
HAVING <fcount>>=SUPPORT
```

donde :

La primitiva EQUIKEEP ON <CondiciónValoresAtributos > se utilizaría en el caso de conocer en un proceso anterior los itemsets frecuentes de un tamaño menor al especificado en la cláusula RANGE <numero1> UNTIL <numero2>. EQUIKEEP ON hace más eficiente el proceso de generar los diferentes itemsets con la primitiva ASSOCIATOR, al hacer nulos los itemsets no frecuentes.

Ejemplo 4.9. Sea la tabla TRANSACCION (TID, ITEM1, ITEM2, ITEM3). Descubrir las Reglas de Asociación unidimensionales con un soporte mínimo de 2 y una confianza

mínima de 3 y almacenarlas en la tabla ReglasAsociacion. Se supone que los itemsets frecuentes de tamaño 1 son {i2}, {i3}, {i4}. El tamaño máximo de itemsets frecuentes en este caso es de 3 por el número de atributos del mismo tipo.

La sentencia SQL que genera las reglas de asociación unidimensionales es:

```
DESCRIBE UNIDIMENSIONAL ASSOCIATION RULES
FROM ItemsFrecuentes INTO ReglasAsociacion
WITH SUPPORT 2 AND CONFIDENCE 3
AS   SELECT item1, item2, item3, count(*) AS soporte INTO ItemsFrecuentes
     FROM transaccion
     EQUIKEEP ON item1 IN (i2,i3,i4) OR item2 IN (i2,i3,i4) OR item3 IN
     (i2,i3,i4)
     ASSOCIATOR RANGE 2 UNTIL 3
     GROUP BY item1, item2, item3 HAVING count(*)>=2
```

En sentencias SQL independientes:

```
SELECT item1, item2, item3, count(*) AS soporte INTO ItemsFrecuentes
FROM transaccion
EQUIKEEP ON item1 IN (i2,i3,i4) OR item2 IN (i2,i3,i4) OR item3 IN (i2,i3,i4)
ASSOCIATOR RANGE 2 UNTIL 3
GROUP BY item1, item2, item3 HAVING count(*)>=2

DESCRIBE UNIDIMENSIONAL ASSOCIATION RULES
FROM ItemsFrecuentes INTO ReglasAsociacion
WITH SUPPORT 2 AND CONFIDENCE 3
```

El proceso de generar reglas de asociación unidimensionales utilizando una tabla de datos de modelo simple se debe realizar en sentencias independientes el cálculo de los itemsets frecuentes y el de producir reglas de asociación. Ejemplo:

```
DESCRIBE UNIDIMENSIONAL ASSOCIATION RULES
FROM tran23 INTO ReglasAsociacion
WITH SUPPORT 2 AND CONFIDENCE 50
```

**Tabla 4.21. Operador Describe Association Rules sobre la tabla TRAN23**

| ID_ITEM1 | ID_ITEM2 | ID_ITEM3 | REGLA | IMPLICA |
|---|---|---|---|---|
| i2 | null | null | 1 | A |
| null | i3 | null | 1 | C |
| null | i3 | null | 2 | A |
| i2 | null | null | 2 | C |
| i2 | null | null | 3 | A |

| | | | | |
|---|---|---|---|---|
| null | null | i4 | 3 | C |
| null | null | i4 | 4 | A |
| i2 | null | null | 4 | C |
| i2 | null | null | 5 | A |
| null | i3 | i4 | 5 | C |
| null | i3 | null | 6 | A |
| i2 | null | i4 | 6 | C |
| null | null | i4 | 7 | A |
| i2 | i3 | null | 7 | C |
| i2 | i3 | null | 8 | A |
| null | null | i4 | 8 | C |
| i2 | null | i4 | 9 | A |
| null | i3 | null | 9 | C |
| null | i3 | i4 | 10 | A |
| i2 | null | null | 10 | C |

**4.7.3 Operador Describe Classification Rules.** Este operador genera todas las reglas de clasificación a partir de una tabla de datos.

La sintaxis de este operador es la siguiente:

DESCRIBE CLASSIFICATION RULES
FROM <NombreTablaArbol> [INTO <TablaReglasClasificacion>]
USING <ListaAtributosCondicion> CLASS <AtributoClase>
[AS <SubconsultaConstrucciónArbol>]

<ListaAtributosCondicion>:=<ListaAtributos><Atributo>
<ListaAtributos>:=<Atributo> | <ListaAtributos>
<AtributoClase>:=<Atributo>
<SubconsultaConstruccionArbol>:=<SELECT FROM WHERE MATE BY GROUP BY>

donde:

La cláusula DESCRIBE CLASSIFICATION RULES FROM <NombreTablaArbol> especifica la tabla <NombreTablaArbol> donde se encuentran los datos necesarios para la construcción del árbol de decisión que servirá de base para la generación de reglas de clasificación.

La cláusula opcional INTO <TablaReglasClasificcion> permite almacenar en una tabla <TablaReglasClasificacion> las reglas de clasificación que se generen, para posteriores consultas.

La cláusula USING <ListaAtributosCondicion> CLASS <AtributoClase> determina en el proceso de clasificación cuales es el conjunto de atributos condición <ListaAtributosCondicion> y cual el atributo clase <AtributoClase>.

La cláusula opcional [AS <SubconsultaConstrucciónArbol>] permite especificar conjuntamente con la cláusula DESCRIBE, la subconsulta <SubconsultaConstruccionArbol> que permite calcular la entropía y ganancia utilizando las funciones agregadas ENTRO() y GAIN() conjuntamente con la primitiva MATE BY para construir el árbol de decisión. Para efectos de optimización del operador DECRIBE CLASSIFICATION RULES, la subconsulta se ejecutaría primero y luego se generarían las reglas de clasificación. En el caso de que esta cláusula no se especifique, los procesos de construcción del árbol de decisión y el de generación de reglas se realizarían de manera independiente. Primero se construiría el árbol mediante una sentencia <SELECT FROM WHERE MATE BY GROUP BY> y posteriormente se generarían las reglas de Clasificación. Ejemplo:

DESCRIBE CLASSIFICATION RULES
FROM sintomas INTO ReglasClasificacion
USING d_cabeza, d_muscular, temperatura CLASS gripa

**Tabla 4.22. Operador Describe Classification Rules sobre la tabla SINTOMAS**

| REGLA | ATRIBUTO | VALOR |
|---|---|---|
| 1 | TEMPERATURA | Alta |
| 1 | GRIPA | Si |
| 2 | TEMPERATURA | Media |
| 2 | GRIPA | Si |
| 3 | D_MUSCULAR | Si |
| 3 | TEMPERATURA | Normal |
| 3 | GRIPA | Si |
| 4 | D_MUSCULAR | No |
| 4 | TEMPERATURA | Normal |
| 4 | GRIPA | No |

## 4.8 CONCLUSIONES

Se ha propuesto un nuevo método para integrar las tareas de asociación y clasificación al interior de un SGBD compuesto por tres fases: en la primera, se extiende el álgebra relacional con nuevos operadores para asociación y clasificación, los cuales ejecutan las tareas más costosas de estos procesos. En la segunda, se extiende el lenguaje SQL con nuevas primitivas que se expresan en la cláusula SQL SELECT y en la tercera, se definen nuevos operadores SQL para la generación de reglas de asociación y clasificación.

La estrategia de implementar los nuevos operadores algebraicos como primitivas SQL dentro de la cláusula SELECT, permite que su optimización se limite a determinar las reglas de ejecución de estos y a su compilación por parte del Parser o analizador sintáctico del SGBD. Proceso que se ha desarrollado.

# 5. IMPLEMENTACIÓN DE LOS OPERADORES DE ASOCIACIÓN Y CLASIFICACIÓN

En el presente capítulo se presenta la implementación de los operadores Associator Range y EquiKeep On de Asociación y el operador Mate By de Clasificación, bajo el mismo lenguaje y acoplado a las características de un operador de Postgres.

Para dar una mejor comprensión de lo realizado en el código fuente de Postgres, primero se presenta una breve recopilación y descripción de la estructura general de un operador en Postgres para luego tratar sobre la adición al Manejador de Bases de Datos Postgres de los Operadores de Asociación y Clasificación.

## 5.1 CLASIFICACIÓN DE OPERADORES EN POSTGRES

Con una consulta realizada en Postgres, 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, el cual es pasado al Executor y procesado por éste.

Si el plan entregado por el Planner/Optimizer al Executor contiene un solo nodo, 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 este compuesto por varios nodos, se denomina "Operador Complejo", ejemplo: Group by, MergeJoin, HashJoin, Agg, NestLoop, Hash.

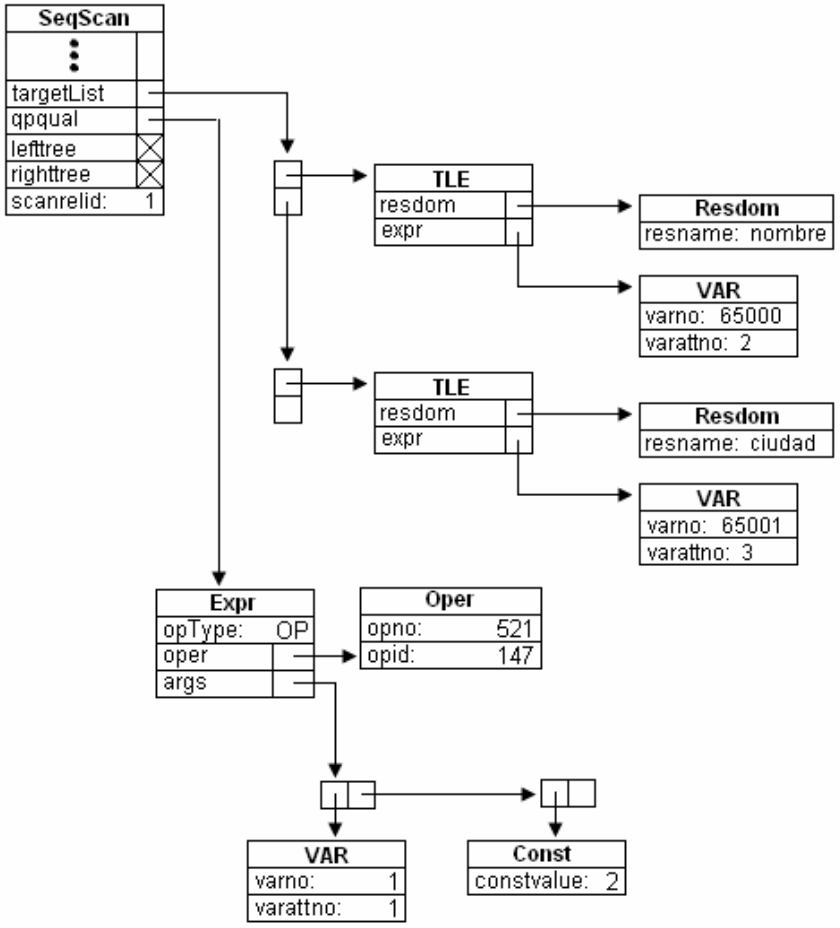
**5.1.1 Ejemplo de un Operador Sencillo.** El Operador SeqScan recorre secuencialmente una tabla en una dirección determinada. Este es un Operador Sencillo debido a que Postgres lo representa con un solo nodo en el plan de ejecución. Este operador al igual que el resto de los Operadores de Postgres, tiene tres rutinas de interfaz asociadas a cada una 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.

Para la Tabla Vendedores (Id, Nombre, Ciudad) se realiza la siguiente consulta para observar el nodo formado por el Operador Sencillo (Nodo SeqScan). (Ver Figura 5.1).

*Select Nombre, Ciudad*
*From Vendedores*
*Where Id >2*;

**Figura 5.1. Plan de ejecución de un Operador Sencillo (SeqScan)**




En la Figura 5.1 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

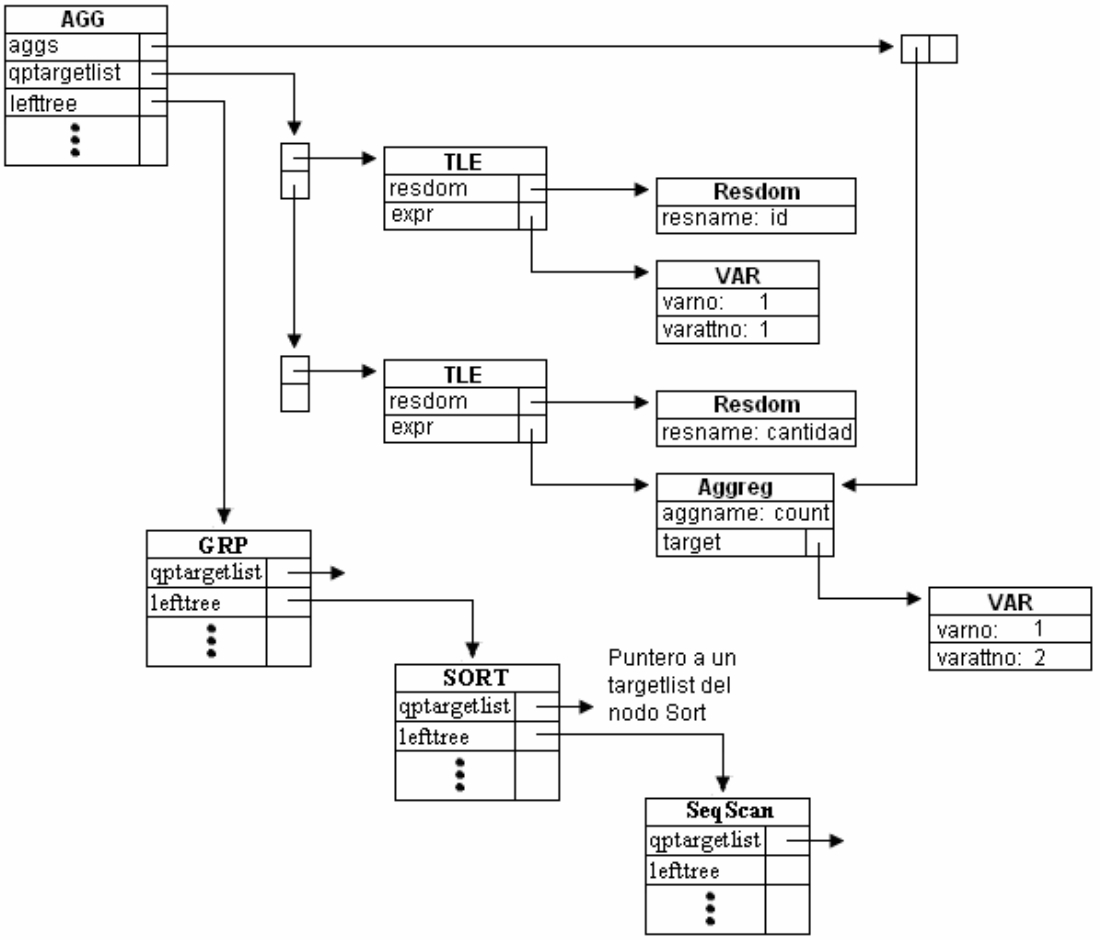
Postgres. El campo qpqual contiene un apuntador a un nodo Expr, en el cual se encuentran presente tres campos, el primero opType, contiene el operador relacional presente en la consulta (>,<,=,…), el segundo es un apuntador a un nodo de tipo Oper, el cual contiene el identificador asignado al operador 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 Ejemplo de 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 asumen 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.

Para la tabla Ventas (Id, Cantidad, Valor), se realiza la siguiente consulta para observar el plan de ejecución formado por el Operador Complejo (Nodo Agg). (Ver Figura 5.2).

*Select Id, count(cantidad)*
*From Ventas*
*Group by Id*;

**Figura 5.2. Plan de ejecución de un Operador Complejo (Agg)**




En la Figura 5.2, 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.

## 5.2 ESTRUCTURA GENERAL DE LOS OPERADORES

Postgres clasifica los Operadores según su tarea (agrupar, ordenar, contar, entre otros), estos se encuentran en 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.

**5.2.1 T.D.A. (Tipo de Dato Abstracto de un nodo).** 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 5.3 se muestra el T.D.A. de un nodo SeqScan.

**Figura 5.3. T.D.A de un nodo SeqScan**

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

En el T.D.A de la Figura 5.3 podemos identificar tres parámetros principales:

**5.2.1.1 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 5.4 se muestra la definición de esta estructura.

**Figura 5.4. T.D.A de un Plan de Ejecución**

```
typedef struct Plan
{
        NodeTag         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             nParamExec;
}Plan;
```

- type (NodeTag): Representa el tipo del Nodo. Cada nodo tiene un tipo o identificador, el cual 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 como: el conjunto de relaciones que intervienen en la consulta, bloques de memoria disponibles, dirección de la búsqueda, entre otros. Este campo es igual para todos los nodos de un mismo plan de ejecución.
- righttree, 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.

**5.2.1.2 Estado de un nodo.** El estado contiene la información necesaria para retornar la próxima 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 CommonState incluida en el archivo *.../src/include/nodes/execnodes.h*. La Figura 5.5 presenta la definición de la estructura CommonState.

**Figura 5.5. T.D.A del estado general de un nodo**

```
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;
```

Sus principales campos o propiedades 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 (omitido en versiones posteriores).
- cs_OuterTupleSlot (TupleTableSlot) (Tupla resultante, sin proyectar): Apuntador a la tupla 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_TupleSlot. 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, el cual 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 5.6), 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.

**Figura 5.6. T.D.A del estado del nodo SeqScan**

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

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

**5.2.2 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/*.

## 5.3 ADICIÓN DEL OPERADOR ASSOCIATOR RANGE

La idea básica de la implementación es modificar las estructuras, funciones y crear nuevos nodos en las etapas que componen la capa intermedia de la arquitectura de Postgres así:

- La etapa Parser ha sido modificada 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 la 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 las siguientes secciones se describe con más detalle, los cambios hechos a cada etapa.

**5.3.1 Etapa Parser.** El análisis léxico no se ha modificado pues no se ha realizado la introducción de nuevos símbolos, siendo innecesario alterar esta etapa.

Para el análisis sintáctico fue necesario introducir cambios y nuevo código en las funciones o estructuras de los siguientes archivos. (Tenga en cuenta que solamente se presenta las partes relevantes de código afectado en lugar de las funciones completas. Cada línea de código agregada será marcada por un '+' al principio de la línea y cada línea de código modificada será marcada por un '!' a través de los siguientes listados de código).

- 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 5.7.

**Figura 5.7. Nuevas reglas de Producción para Associator**

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

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

Ahora, en la regla simple_select se establece el punto para activar las nuevas reglas de producción (Figura 5.8).

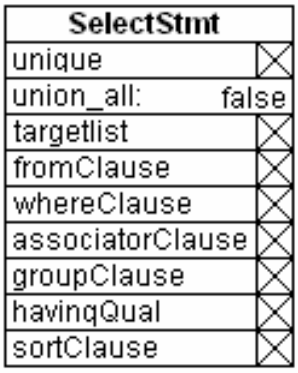
**Figura 5.8. Regla simple_select modificada para Associator**

```
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;
        }
```

- Para recibir y almacenar los valores de rango inicial y final del operador Asociator, fue necesario agregar una lista a la estructura SelectStmt (*…/src/include/nodes/parsenodes.h*). La Figura 5.9 presenta la estructura SelectStmt modificada.

**Figura 5.9. SelectStmt modificada para Associator**

```
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;
```




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 SelectStmt a un nodo Query, se creó la función transformAssociatorClause, cuya definición está en el archivo *…/src/include/parser/parser_clause.h* y se implementó físicamente en *…/src/backend/parser/parser_clause.c* (Figura 5.10).

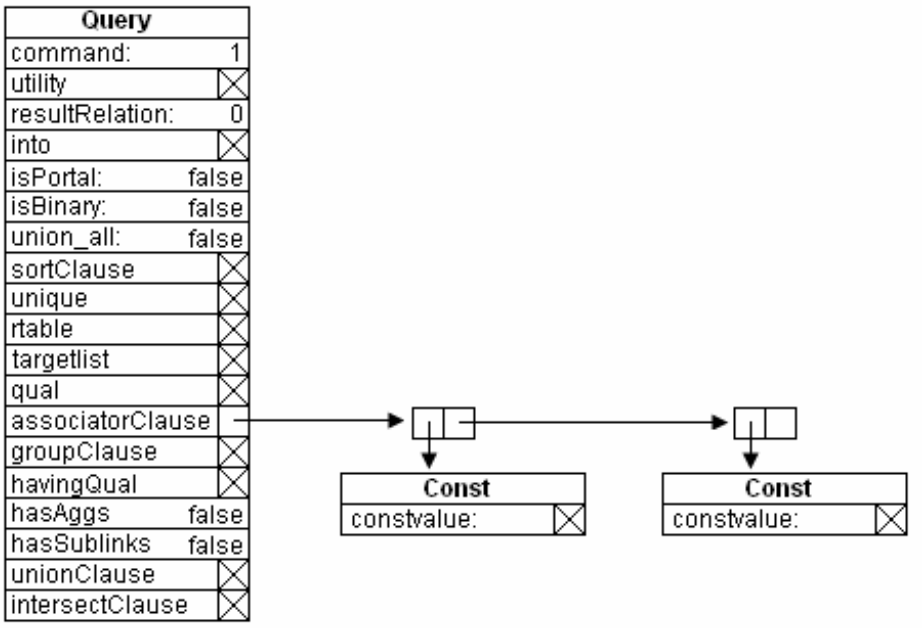
**Figura 5.10. Función transformAssociatorClause**

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

+     return alist;
+ }
```

- Se modificó el nodo Query (*../src/include/nodes/parsenodes.h*) con el fin de que pueda recibir y almacenar los datos de la estructura SelectStmt que pertenecen al nuevo operador. La Figura 5.11 presenta el nodo Query modificado.

**Figura 5.11. Query modificado para Associator**

```
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;
```




- 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.

**5.3.2 Etapa Rewrite.** Puesto que este proyecto no contempla la implementación de las primitivas Data Mining como parte de la definición de vistas, ni reglas de transformación a partir de estas, no se realizó ninguna modificación a los módulos que pertenecen a esta etapa.

**5.3.3 Etapa 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 poder 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 5.12 presenta el nuevo nodo para Associator.

**Figura 5.12. Nodo Associator**

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

- Para identificar la nueva estructura AssociatorState primero se adiciona la etiqueta T_AssociatorState 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 5.13. 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.

**Figura 5.13. AssociatorState**

```
+    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;
```

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 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): 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 (void): 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 5.14).

**Figura 5.14. Función make_associator**

```
+       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;
+       }
```

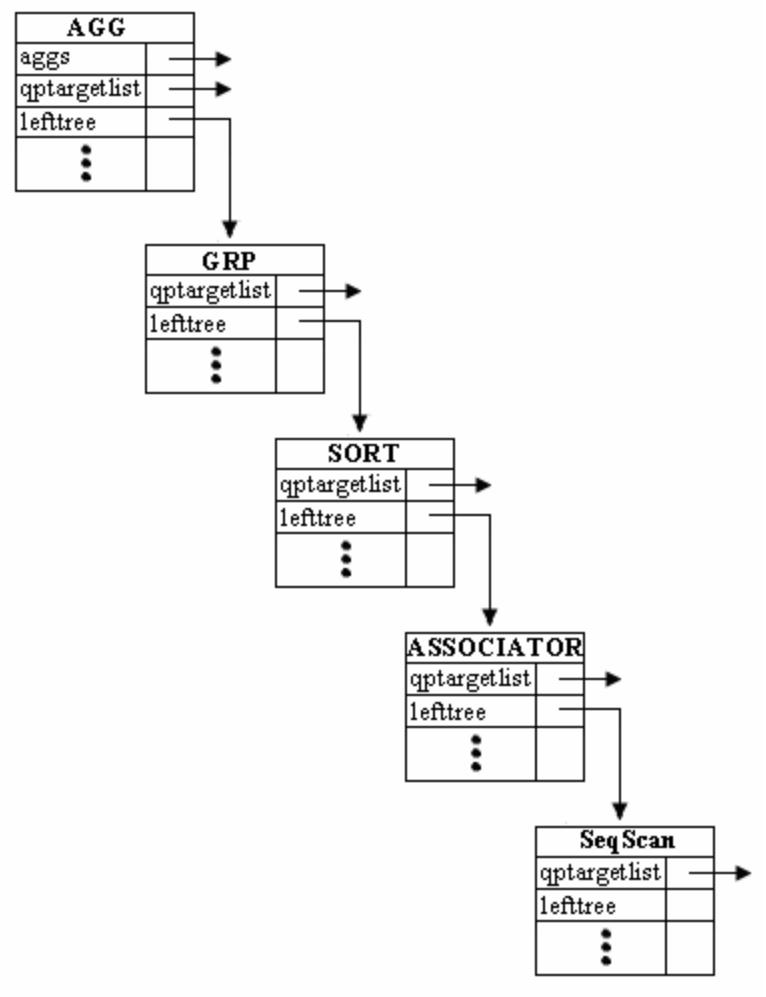
- La función grouping_planner del archivo …*/src/backend/optimizer/plan/planner.c* es la encargada de asignar el orden de los nodos en el plan de ejecución, por tal razón es necesario incluir en este punto, una referencia a la función make_associator como lo muestra la Figura 5.15.

**Figura 5.15. Función grouping_planner modificado para Associator**

```
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);
+       }
        ...
}
```

- 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 5.16 presenta el plan de ejecución de la primitiva Associator Range en un operador complejo.

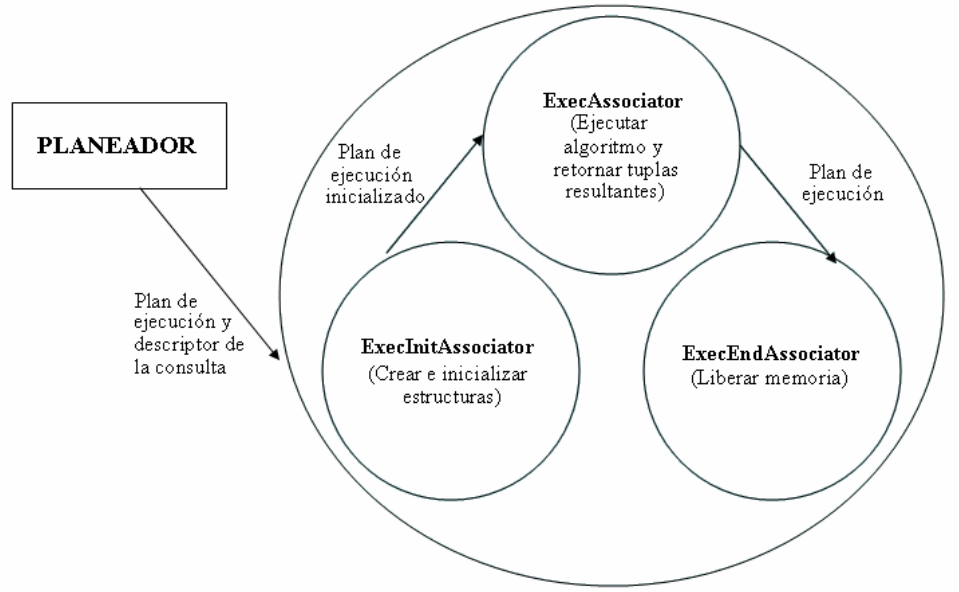
**Figura 5.16. Plan de Ejecución de Associator Range**




**5.3.4 Etapa 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 Executor: 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 crea el archivo *…/src/include/executor/nodeAssociator.h* y se crea el archivo *…/src/backend/executor/nodeAssociator.c* para la implementación de las mismas. La Figura 5.17 presenta las funciones de manipulación del nodo Associator.

**Figura 5.17. 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 InitPlan, 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 5.18).

**Figura 5.18. Función ExecInitNode modificado para Associator**

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

- 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 5.19).

**Figura 5.19. Función ExecProcNode modificado para Associator**

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

- 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 5.20).

**Figura 5.20. Función ExecEndNode modificado para Associator**

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

El operador *Assorow* trabaja de manera similar que *Associator*, es decir, genera por cada tupla de una relación todos sus posibles subconjuntos (itemsets) de diferente tamaño, pero sin hacer nulos los atributos que no forman el itemset. Los valores nulos siempre estarán en los atributos de la parte derecha de cada tupla, que por el tamaño del itemset, no se ocupan. Por tal razón, la manera de implementación de Assorow conserva las mismas características de Associator y por tanto no se presentan detalles de su implementación en este documento.

## 5.4 ADICIÓN DEL OPERADOR EQUIKEEP ON

Para la implementación del operador Equikeep On, se deben modificar 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 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.

En las siguientes secciones se describe con más detalle, los cambios hechos a cada etapa.

**5.4.1 Etapa 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 5.21.

**Figura 5.21. Nuevas reglas de Producción para Equikeep**

```
+       equikeep_clause:
+               EQUIKEEP keep_list      { $$ = $2; }
+               | /*EMPTY*/             { $$ = NIL; }
+               ;
+       keep_list:
+               a_expr                  { $$ = makeListi1($1); }
+               | keep_list ',' a_expr  { $$ = lappendi($1, $3); }
+               ;
```

Ahora, en la regla simple_select se establece el punto para activar las nuevas reglas de producción (Figura 5.22).

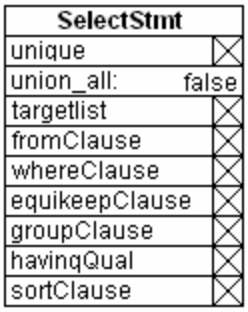
**Figura 5.22. Regla simple_select modificada para Equikeep**

```
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;
                }
```

- Para recibir y almacenar los valores al conjunto de expresiones lógicas del operador Equikeep, se agregó una lista a la estructura SelectStmt (*…/src/include/nodes/parsenodes.h*). La Figura 5.23 presenta la estructura SelectStmt modificada.

**Figura 5.23. SelectStmt modificada para Equikeep**

```
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;
```




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 SelectStmt a un nodo Query, se creó la función transformEquikeepClause, cuya definición está en el archivo *…/src/include/parser/parser_clause.h* y se implementó físicamente en *…/src/backend/parser/parser_clause.c* (Figura 5.24).

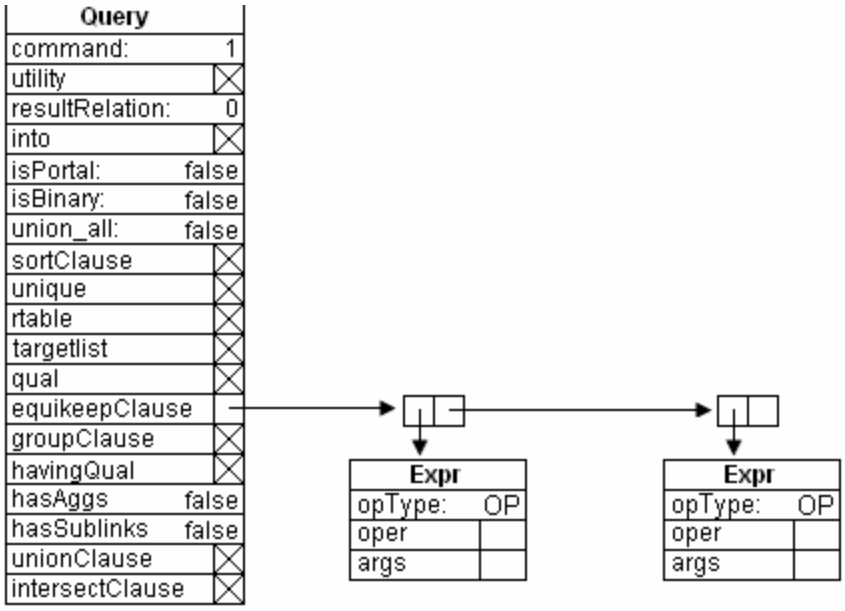
**Figura 5.24. Función transformEquikeepClause**

```
+	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;
+	}
```

- Se modificó el nodo Query (*../src/include/nodes/parsenodes.h*) con el fin de que pueda recibir y almacenar los datos de la estructura SelectStmt que pertenecen al nuevo operador. La Figura 5.25 presenta el nodo Query modificado.

**Figura 5.25. Query modificado para Equikeep**

```
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;
```




- 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.

**5.4.2 Etapa Rewrite.** Puesto que este proyecto no contempla la implementación de las primitivas Data Mining como parte de la definición de vistas, ni reglas de transformación a partir de estas, no se realizó ninguna modificación a los módulos que pertenecen a esta etapa.

**5.4.3 Etapa Planner/Optimizer.** Para adicionar el operador Equikeep 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_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 5.26 presenta el nuevo nodo para Equikeep.

**Figura 5.26. Nodo Equikeep**

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

- Para identificar la nueva estructura EquikeepState primero se adiciona la etiqueta T_EquikeepState 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 5.27. 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.

**Figura 5.27. EquikeepState**

```
+   typedef struct EquiKeepState
+   {
+       CommonScanState csstate;
+       List    *kqual;
+   } 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 5.28).

**Figura 5.28. Función make_equikeep**

```
+   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;
+   }
```

- Antes de unir el nuevo nodo al plan de ejecución, se necesitó transformar todas las expresiones lógicas de 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/optimizer/plan/planner.c* como lo muestra la Figura 5.29.

**Figura 5.29. Función Subquery_planner modificado para 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;
+
        ...
}
```

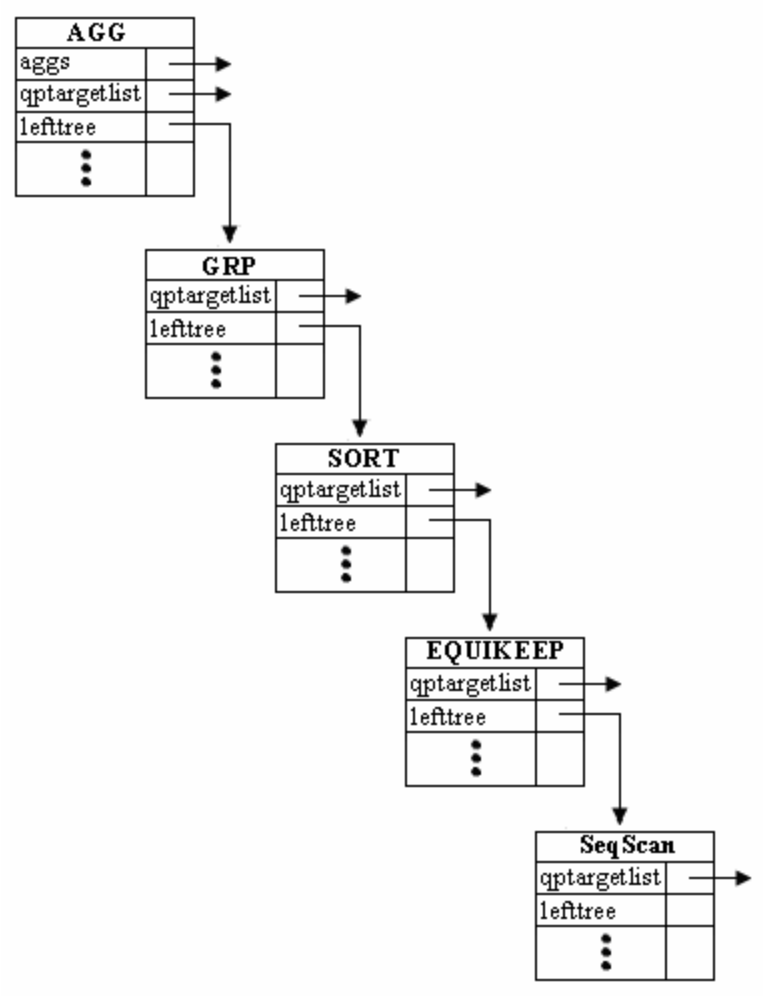
- La función grouping_planner del archivo …*/src/backend/optimizer/plan/planner.c* es la encargada de asignar el orden de los nodos en el plan de ejecución, por tal razón es necesario incluir en este punto, una referencia a la función make_equikeep, como lo muestra la Figura 5.30.

**Figura 5.30. Función grouping_planner modificado para Equikeep**

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

- Por último 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 realice los ajustes necesarios para que el Executor pueda trabajar adecuadamente. La Figura 5.31 presenta el plan de ejecución de la primitiva Equikeep On en un operador complejo.

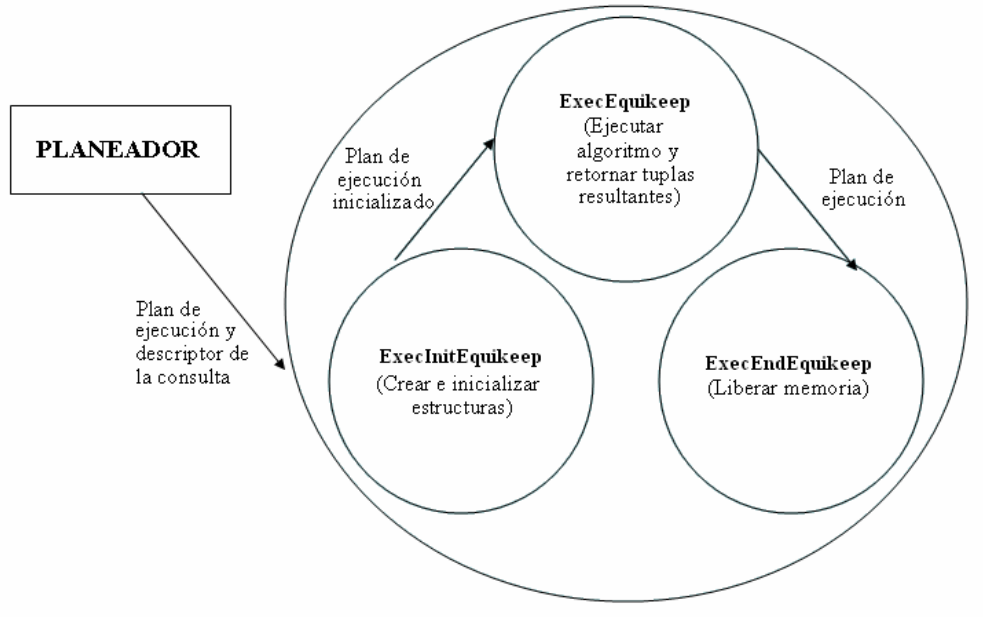
**Figura 5.31. Plan de Ejecución de Equikeep On**




**5.4.4 Etapa Executor.** Al igual que Associator, para Equikeep se necesita definir e implementar las rutinas de manipulación del nodo (ExecInitEquikeep, ExecEquikeep y ExecEndEquikeep), para lo cual se realizan 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 5.32.

**Figura 5.32. 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 5.33)

**Figura 5.33. Función ExecInitNode modificado para Equikeep**

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

- 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 Equikeep (Figura 5.34).

**Figura 5.34. Función ExecProcNode modificado para Equikeep**

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

- 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 5.35).

**Figura 5.35. Función ExecEndNode modificado para Equikeep**

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

## 5.5 ADICIÓN DEL OPERADOR MATE BY

Para la implementación del operador Mate by, se deben modificar 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 ha sido modificado para generar por cada una de las tuplas, todas las 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.

En las siguientes secciones se describe con más detalle, los cambios hechos a cada etapa.

**5.5.1 Etapa 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 archivos que se mencionan adelante.

- 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 5.36.

**Figura 5.36. Nuevas reglas de Producción para Mate**

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

Ahora, en la regla simple_select se establece el punto para activar las nuevas reglas de producción (Figura 5.37).

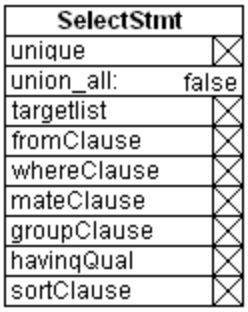
**Figura 5.37. Regla simple_select modificada para Mate**

```
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;
                        }
```

- Para recibir y almacenar los atributos condición y el atributo clase del operador Mate, se agregó una lista a la estructura SelectStmt (*…/src/include/nodes/parsenodes.h*). La Figura 5.38 presenta la estructura SelectStmt modificada.

**Figura 5.38. SelectStmt modificada para Mate**

```
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;
```




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 SelectStmt a un nodo Query, se creó la función transformMateClause, cuya definición está en el archivo *…/src/include/parser/parser_clause.h* y se implementó físicamente en *…/src/backend/parser/parser_clause.c* (Figura 5.39).

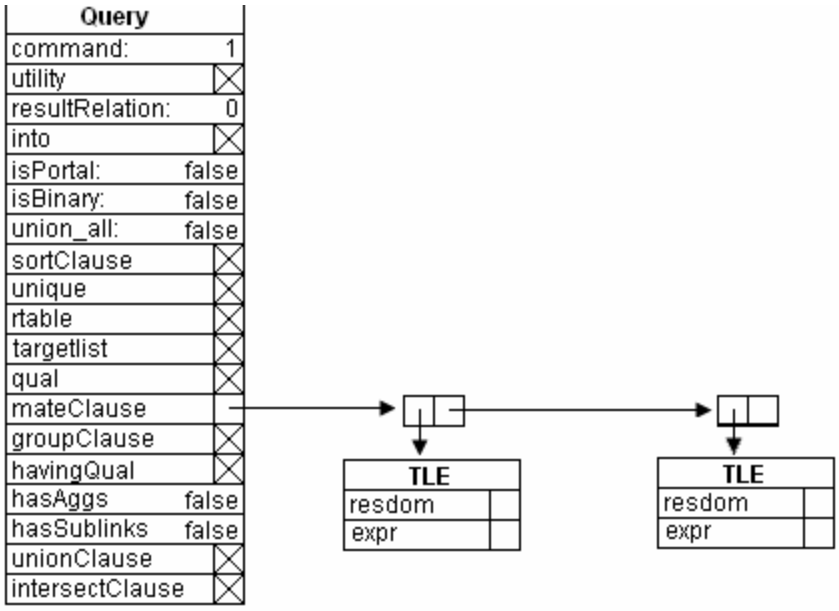
**Figura 5.39. Función transformMateClause**

```
+	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;
+	}
```

- Se modificó el nodo Query (*…/src/include/nodes/parsenodes.h*) con el fin de que pueda recibir y almacenar los datos de la estructura SelectStmt que pertenecen al nuevo operador. La Figura 5.40 presenta el nodo Query modificado.

**Figura 5.40. Query modificado para Mate**

```
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;
```




- 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 transformMateClause.

**5.5.2 Etapa Rewrite.** Puesto que este proyecto no contempla la implementación de las primitivas Data Mining como parte de la definición de vistas, ni reglas de transformación a partir de estas, no se realizó ninguna modificación a los módulos que pertenecen a esta etapa.

**5.5.3 Etapa 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 5.41 presenta el nuevo nodo para Mate.

**Figura 5.41. Nodo Mate**

```
+   typedef struct Mate
+   {
+       Plan        plan;
+       List        *mateList;
+       MateState   *matestate;
+   } 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 5.42. 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 5.42. MateState**

```
+    typedef struct MateState
+    {
+        CommonScanState csstate;
+        List        *mateList;
+        bool        pedir_tupla;
+        bool        retornar_tupla;
+        int         num_tuplas;
+        int         out_tuplas;
+        TupleTableSlot    *tupla;
+        void              *tuplestorestate;
+    } 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 (void): 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 5.43).

**Figura 5.43. Función make_mate**

```
+       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;
+       }
```

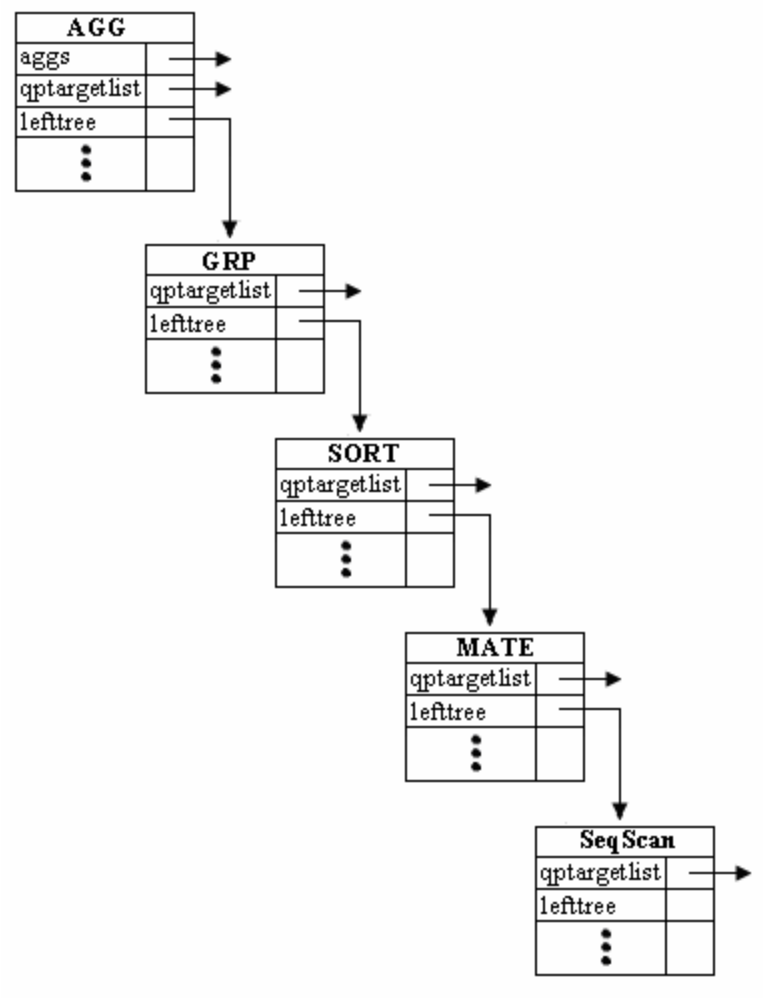
- La función grouping_planner del archivo *…/src/backend/optimizer/plan/planner.c* es la encargada de asignar el orden de los nodos en el plan de ejecución, por tal razón es necesario incluir en este punto, una referencia a la función make_mate, como lo muestra la Figura 5.44.

**Figura 5.44. Función grouping_planner modificado para 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);
+               }
…
}
```

- Por último 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 5.45 presenta el plan de ejecución de la primitiva Mate By en un operador complejo.

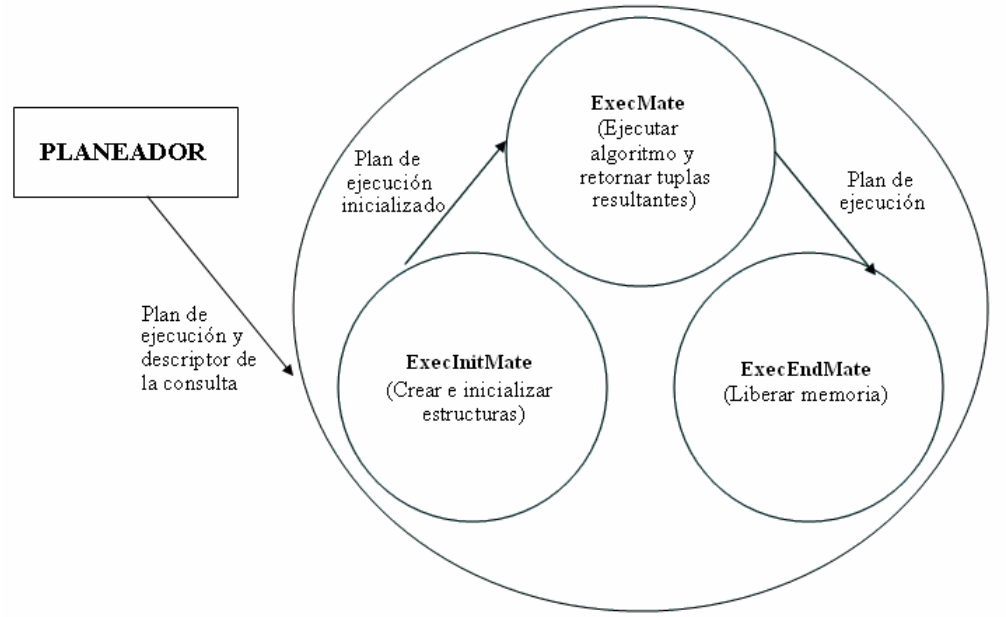
**Figura 5.45. Plan de Ejecución de Mate By**




**5.5.4 Etapa Executor.** Al igual que 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 5.46.

**Figura 5.46. Funciones de manipulación del nodo Mate**




- 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 5.47)

**Figura 5.47. Función ExecInitNode modificado para Mate**

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

- 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 5.48).

**Figura 5.48. Función ExecProcNode modificado para Mate**

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

- 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 5.49).

**Figura 5.49. Función ExecEndNode modificado para Mate**

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

## 5.6 ADICION DE FUNCIONES DEFINIDAS POR EL USUARIO (FDU)

### 5.6.1 Implementación de las funciones entro() y gain() para Clasificación, y la función describe_association_rules() para Asociación.

**5.6.1.1 Programación en el Servidor**. PostgreSQL le proporciona al usuario la capacidad de diseñar e implementar funciones de propósito especializado o general. Dichas funciones se escriben en un lenguaje de programación anfitrión C y en un lenguaje basado en SQL como PL/pgSQL.

Debido a que Postgresql (al menos hasta la versión 7.3.4) no ofrece soporte para incluir coherentemente funciones externas como funciones agregadas que el backend pueda reconocer y tratar directa y eficientemente, incluye entre sus directorios uno por defecto para que el usuario pueda una vez creadas sus funciones establecer aquí la jerarquía necesaria para su correcto funcionamiento, aunque bien se puede evitar este formalismo es recomendable hacer uso de él. Este es el directorio Contrib ubicado en *./postgresql/contrib.*

Para que una FDU pueda ejecutarse es necesario además del código fuente, construir los archivos *.sql* donde se especifica la instrucción de instalación para Postgres y las fuentes.

**5.6.1.2 Adición de la Función Definida por el Usuario entro() para Clasificación.** Los archivos que contienen las fuentes de la función entro() se ubican en el directorio: *./postgresql_7.3.4/contrib/kdd/clasificacion/* que corresponde a:

entro.sql: contiene las instrucciones en PL/pgSQL necesarias para la ejecución de entro().

Es necesario para el funcionamiento de una función escrita en un lenguaje procedural basado en SQL como PL/pgSQL, registrar el lenguaje para su uso en una base de datos específica. Esto se hace accediendo como usuario postgres (o un usuario con privilegios equivalentes) y dando desde el prompt ($) la instrucción:

*CREATE LANGUAJE PLPGSQL <<dbname>>*

Donde *dbname* es el nombre de la base de datos específica.

La creación de la función se realiza accediendo a la base de datos, ejecutando los programas monitores de Postgres como psql (que permite introducir, editar y ejecutar comandos SQL interactivamente). Luego, se utiliza el comando de psql "\i" para leer peticiones a partir del archivo *\i /ruta_archivo/nombre_archivo.*

Así se ejecuta la instrucción *\i /postgresql_7.3.4/contrib/kdd/clasificacion/entro.sql.*

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

A partir de este momento se puede invocar a la función entro() con los parámetros requeridos, que como precondición debió incluir el uso de la primitiva MATE en una consulta de operador complejo, la cual hace uso de la función agregada count(), la opción INTO y la cláusula GROUP BY. El resultado de esta consulta es una tabla con el formato adecuado, que puede utilizarse como parámetro de *entro('nombre_tabla')*.

**5.6.1.2.1 Funcionamiento de entro().** 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 sea necesario 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.

Por ejemplo, sobre la tabla apariencia (Figura 5.50), el resultado de la sentencia:

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

Se muestra en la tabla de la Figura 5.51. Al aplicar la función entro() sobre la tabla mateapariencia se obtiene las tablas 'mate_values' (Figura 5.52) y 'mate_entro' (Figura 5.53).

**Figura 5.50. Tabla de clasificación apariencia**

| 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 5.51. Tabla de clasificación mateapariencia**

| tamano | cabello | ojos | cutis | count |
|---|---|---|---|---|
| alto | claro | azul | claro | 1 |
| alto | claro | cafe | oscuro | 1 |
| alto | claro | | claro | 1 |
| alto | claro | | oscuro | 1 |
| alto | oscuro | azul | claro | 1 |
| alto | oscuro | cafe | oscuro | 1 |
| alto | oscuro | | claro | 1 |
| alto | oscuro | | oscuro | 1 |
| alto | rojo | azul | claro | 1 |
| alto | rojo | | claro | 1 |
| alto | | azul | claro | 3 |
| alto | | cafe | oscuro | 2 |
| alto | | | claro | 3 |
| alto | | | oscuro | 2 |
| corto | claro | azul | oscuro | 1 |

...
...

**Figura 5.52. Tabla de clasificación mate_values para apariencia**

| 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 5.53. Tabla de clasificación mate_entro para apariencia**

| 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 |

...
...

**5.6.1.3 Adición de la Función Definida por el Usuario gain() para Clasificación.** Los archivos que contienen las fuentes de la función *gain()* se ubican en el directorio: *./postgresql_7.3.4/contrib/kdd/clasificacion/* 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/* 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.sql*

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

A partir de este momento ya es posible invocar a la función gain() haciendo uso de los parámetros requeridos, que como precondición se debió incluir el uso de la primitiva MATE en una consulta de operador complejo, lo cual hace uso de la función agregada count(), la opción INTO, la cláusula GROUP BY, y la función entro().

**5.6.1.3.1 Funcionamiento de Gain().** El parámetro de entrada es un dato de tipo VARCHAR que contiene el nombre de la tabla, que previamente se debió tratar con la función entro(), es decir una tabla con el formato 'mate_entro'. Dentro de la cláusula SELECT, gain() tiene la siguiente sintaxis:

SELECT * FROM GAIN('<*mate_entro*>')

Internamente se extrae de la tabla 'mate_values' cuantos atributos posee mate_entro, y el número de valores posibles para cada atributo existente.

Se extrae los valores de entropía y se calcula la ganancia de información para establecer que atributo se convierte en la raíz del árbol de clasificación.

A partir de aquí se recorre la tabla para buscar registros cuyos atributos coincidan con los valores existentes en el árbol para calcular las nuevas ganancias de información; se 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). En este momento se ha construido en memoria principal el árbol de decisión como se estipula en el capítulo 2.

Las subrutinas de TablaArbol() y TablaRules() recorren el árbol en memoria y a partir de él construyen dos tablas físicas, que contienen información relevante para construir 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. Son independientes entre si y existen físicamente, por tanto son persistentes.

Estas tablas pueden convertirse a valores originales utilizando la información contenida en 'mate_values' creada previamente. Los diferentes valores del campo atributo de 'mate_values' se referencian secuencialmente con valores enteros iniciando en 0.

La estructura de las tablas 'tclases' y 'trulesclases' se muestra en las Figuras 5.54 y 5.55 respectivamente. La tabla 'tclases' especifica las relaciones entre los nodos del árbol, y los valores útiles. Todos 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 valor de ganancia el atributo clase pudo 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.

**Figura 5.54. Tabla tclases**

| 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 5.55. Tabla trulesclases**

| 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 |

Por ejemplo, de la tabla trulesclases (Figura 5.55) la primera regla se interpreta como:
Si atributo[0] = valor[0] y si atributo[2] = valor[0] entonces atributo[clase] = valor[0]
Y significa:
Si tamaño = alto y ojos = azul entonces cutis = claro.

La cual es una regla de clasificación valida para la tabla original de la Figura 5.50.

**5.6.1.4 Adición de la Función Definida por el Usuario describe_association_rules() para Asociación.** Los archivos que contienen las fuentes de la función describe_association_rules() 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 específica 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() haciendo uso de los parámetros requeridos, que como precondición debieron incluir el uso de la primitiva ASSOCIATOR RANGE en una consulta de operador complejo, lo cual hace uso de la función agregada count(), la opción INTO, la cláusula GROUP BY y HAVING.

**5.6.1.4.1 Funcionamiento de describe_association_rules().** Los parámetros de entrada son un dato de tipo VARCHAR que contiene el nombre de la tabla, que previamente se debió tratar con 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>*)

Las reglas son representadas 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.

Por ejemplo, teniendo la tabla transaccion de la Figura 5.56.

**Figura 5.56. Tabla transaccion**

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

El resultado de la sentencia:

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

Se muestra en la tabla de la Figura 5.57.

**Figura 5.57. Tabla assotransaccion**

| 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 |

En la Figura 5.58 se muestra el resultado de la función describe_association_rules() para la tabla assotransaccion con un tamaño de regla de 4 y una confianza mínima del 80%.

**Figura 5.58. Tabla describe_association_rules**

| a | b | c | d | n_regla | implica | soporte | confianza |
|---|---|---|---|---|---|---|---|
| - | b1 | - | - | 2 | A | 1 | 100.00 |
| a1 | - | c1 | d1 | 2 | C | 1 | |
| - | - | - | d1 | 4 | A | 1 | 100.00 |
| a1 | b1 | c1 | - | 4 | C | 1 | |
| a1 | b1 | - | - | 5 | A | 1 | 100.00 |
| - | - | c1 | d1 | 5 | C | 1 | |
| a1 | - | - | d1 | 7 | A | 1 | 100.00 |
| - | b1 | c1 | - | 7 | C | 1 | |
| - | b1 | c1 | - | 8 | A | 1 | 100.00 |
| a1 | - | - | d1 | 8 | C | 1 | |
| - | b1 | - | d1 | 9 | A | 1 | 100.00 |
| a1 | - | c1 | - | 9 | C | 2 | |
| - | - | c1 | d1 | 10 | A | 1 | 100.00 |
| a1 | b1 | - | - | 10 | C | 1 | |
| a1 | b1 | c1 | - | 11 | A | 1 | 100.00 |
| - | - | - | d1 | 11 | C | 1 | |
| a1 | b1 | - | d1 | 12 | A | 1 | 100.00 |
| - | - | c1 | - | 12 | C | 2 | |
| a1 | - | c1 | d1 | 13 | A | 1 | 100.00 |
| - | b1 | - | - | 13 | C | 1 | |
| - | b1 | c1 | d1 | 14 | A | 1 | 100.00 |
| a1 | - | - | - | 14 | C | 2 | |
| - | b2 | - | - | 16 | A | 1 | 100.00 |
| a1 | - | c1 | d2 | 16 | C | 1 | |
| - | - | - | d2 | 18 | A | 1 | 100.00 |
| a1 | b2 | c1 | - | 18 | C | 1 | |
| a1 | b2 | - | - | 19 | A | 1 | 100.00 |
| - | - | c1 | d2 | 19 | C | 1 | |
| a1 | - | - | d2 | 21 | A | 1 | 100.00 |
| - | b2 | c1 | - | 21 | C | 1 | |
| - | b2 | c1 | - | 22 | A | 1 | 100.00 |
| a1 | - | - | d2 | 22 | C | 1 | |
| - | b2 | - | d2 | 23 | A | 1 | 100.00 |
| a1 | - | c1 | - | 23 | C | 2 | |
| - | - | c1 | d2 | 24 | A | 1 | 100.00 |
| a1 | b2 | - | - | 24 | C | 1 | |
| a1 | b2 | c1 | - | 25 | A | 1 | 100.00 |
| - | - | - | d2 | 25 | C | 1 | |
| a1 | b2 | - | d2 | 26 | A | 1 | 100.00 |
| - | - | c1 | - | 26 | C | 2 | |
| a1 | - | c1 | d2 | 27 | A | 1 | 100.00 |
| - | b2 | - | - | 27 | C | 1 | |
| - | b2 | c1 | d2 | 28 | A | 1 | 100.00 |
| a1 | - | - | - | 28 | C | 2 | |

De la anterior tabla algunas de las reglas que se puede interpretar 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.

Las instrucciones de instalación de postgresql-kdd y sus funciones de Asociación y Clasificación se presentan en el Anexo E de este documento.

Un ejemplo completo de utilización de las primitivas y funciones de Asociación y Clasificación se presenta en el Anexo D de este documento.

# 6. PRUEBAS Y RESULTADOS

Este capítulo comprende la elaboración de pruebas y el análisis de resultados tanto para las primitivas de Asociación como de Clasificación, con un conjunto de bases de datos construidas para éste fin.

## 6.1 EJECUCION DE PRUEBAS

Para estimar el rendimiento de las primitivas implementadas, se realizaron varias pruebas usando un equipo Pentium IV con 512 MB a 2.8 Ghz, corriendo Red Hat Linux 9.

Para poder aplicar y validar las técnicas de Data Mining en general, y en particular aquellas de Asociación y de Clasificación que constituyen el enfoque del presente trabajo, se necesita un gran volumen de datos. Entonces, para evaluar las primitivas de Asociación, el conjunto de datos utilizados en las pruebas, pertenecen a las transacciones de un supermercado de una de las Cajas de Compensación más importantes del departamento de Nariño (empresa que presta servicios de salud, educación, mercadeo, vivienda, recreación, turismo entre otros) durante un periodo determinado.

Para evaluar la primitiva de Clasificación se utilizó repositorios especializados para éste tipo de algoritmo; se tomaron bases de datos relativamente pequeñas zoo.data (5 Kb, con 101 transacciones) y auto-mpg.data (30 Kb, con 398 transacciones), ambas disponibles en http://www.ics.uci.edu/~mlearn/databases/.

Para medir exactamente los tiempos de ejecución de las diferentes sentencias SQL con los nuevos operadores, se activo la opción \timing de PostgreSQL.

## 6.2 RESULTADOS OBTENIDOS PARA ASOCIACIÓN

El conjunto de datos utilizados en las pruebas pertenecen a las transacciones de un supermercado de una de las Cajas de Compensación más importantes del departamento de Nariño durante un periodo determinado. El tamaño de la primera muestra es de 517.956 transacciones y la segunda muestra es de 1.039.906 transacciones. Con la aplicación de las primitivas de Asociación se pretende analizar las transacciones para intentar descubrir patrones de comportamiento de compra de los clientes en este supermercado.

Es preciso aclarar algunos términos relacionados con las Reglas de Asociación. Así el soporte (s) de una regla se refiere al número de ocurrencias del conjunto de elementos en la base de datos de transacciones, mientras que la confianza (a) es el porcentaje de transacciones que contienen a todos los elementos que componen la regla. Con la aplicación de la función *Describe Association Rules* también implementada en PostgreSQL,

se busca las correlaciones entre las asociaciones generadas por Associator Range y Equikeep a fin de descubrir nuevos patrones y validar los ya encontrados.

En algunas pruebas fue necesario utilizar las 2 primitivas SQL de Asociación: Associator Range y EquiKeep en conjunto con el fin de determinar en que condiciones pueden reducir significativamente el tiempo de generación de los itemset frecuentes. Equikeep permite conservar únicamente los ítems frecuentes en cada transacción, mientras que Associator Range asocia éstos ítems para generar los itemsets frecuentes del tamaño indicado**.**

**6.2.1 Tareas de Preprocesamiento.** La principal tarea en esta etapa, es transformar los archivos log a formato de bases de datos sintéticas. En éstas bases de datos se generan transacciones sintéticas sobre una población simulada de ítems, las cuales representan las compras en el contexto de un supermercado. El modelo del mundo real que se trata de simular es que la gente tiende a comprar al tiempo conjunto de ítems, donde cada conjunto es un potencial large itemset maximal. Para llevar a cabo el proceso de generación debe tenerse en cuenta que una transacción puede contener mas de un large itemset, y su tamaño difiere poco de las demás en la mayoría de los casos. De igual forma, los large itemsets también tienen un tamaño promedio [AgSr94]. Las bases de datos sintéticas manejan los parámetros presentados en la tabla 6.1.

**Tabla 6.1. Parámetros de una base de datos sintética**

| | |
|---|---|
| \|D\| | Número de transacciones. |
| \|T\| | Tamaño promedio de las transacciones. |
| \|I\| | Tamaño promedio de los potenciales large itemsets maximales. |
| \|L\| | Número de potenciales large itemsets maximales. |
| N | Número de Ítems. |

Se realizaron varias tareas de preprocesamiento previas a la aplicación de las primitivas de Asociación sobre el conjunto de datos colectados en los archivos log, como eliminar los productos repetidos en una misma transacción y transformar el conjunto de datos del modelo simple al modelo multicolumna, con el cual operan las primitivas *Associator Range* y *EquiKeep*. El resultado final se almacenó en dos tablas PostgreSQL llamadas T5I3D83K y T10I4D145K, dónde en la primera tabla el tamaño promedio de las transacciones es 5, con tamaño de 3 large itemsets potenciales y un total de 83.720 registros válidos (83K) y para la segunda el tamaño promedio de transacciones es 10, con tamaño de 4 large itemsets potenciales y un total de 144.954 registros validos (145K).

Sobre T5I3D83K y T10I4D145K se aplicó un par de consultas SQL para obtener los itemsets frecuentes tamaño 1 de cada conjunto de datos. Estos ítems permitieron construir la expresión lógica adecuada de la primitiva Equikeep.

La confianza y los soportes mínimos para T5I3D83K y T10I4D145K se establecieron de acuerdo a las características de cada base de datos, con el fin de evitar obtener gran cantidad de itemsets frecuentes que conllevan a la generación de reglas sin importancia.

**6.2.2 Pruebas y Resultados**. La metodología utilizada consistió en aplicar las nuevas primitivas de asociación sobre las bases de datos, y comparar los tiempos de respuesta con la cantidad de registros resultantes, para diferentes soportes mínimos.

Con la aplicación de esta técnica se intenta descubrir todas las asociaciones y correlaciones entre los artículos comprados por los usuarios, ignorando las que tienen un soporte pobre, es decir las que no aparecen en un número suficiente de transacciones.

Como se mencionó en el capítulo 2, el problema de encontrar reglas de asociación se descompone en dos pasos, primero, descubrir los itemsets frecuentes y segundo, usar éstos itemsets para generar las reglas de asociación. Para la evaluación de las pruebas, se consideró únicamente el tiempo de ejecución del primer paso, porque el tiempo de cálculo de reglas de asociación es despreciable en relación al tiempo del primer paso.

La tabla 6.2 muestra los tiempos de ejecución y el número de large Itemsets obtenidos al aplicar en conjunto las primitivas de asociación con diferentes soportes sobre la base de datos T5I3D83K. Los tiempos de ejecución excluyen el preprocesamiento. Se utilizó porcentajes de soporte entre 0.6% y 0.06% y confianza del 20%, ya que para porcentajes menores se obtienen una gran cantidad de reglas no válidas, lo que conlleva a un análisis posterior poco significativo.

**Tabla 6.2. Resultados obtenidos para T5I3D83K con Equikeep y Associator**

| Soporte (%) | Tiempo (Seg.) | Número de Large Itemsets | | |
|---|---|---|---|---|
| | | Total | Tam 2 | Tam 3 |
| 0.6 | 7 | 5 | 5 | 0 |
| 0.3 | 28 | 10 | 10 | 0 |
| 0.1 | 206 | 54 | 41 | 13 |
| 0.06 | 423 | 137 | 106 | 31 |

La tabla 6.3 muestra los tiempos de ejecución y el número de Large Itemsets obtenidos al aplicar únicamente la primitiva Associator Range sobre la misma base de datos.

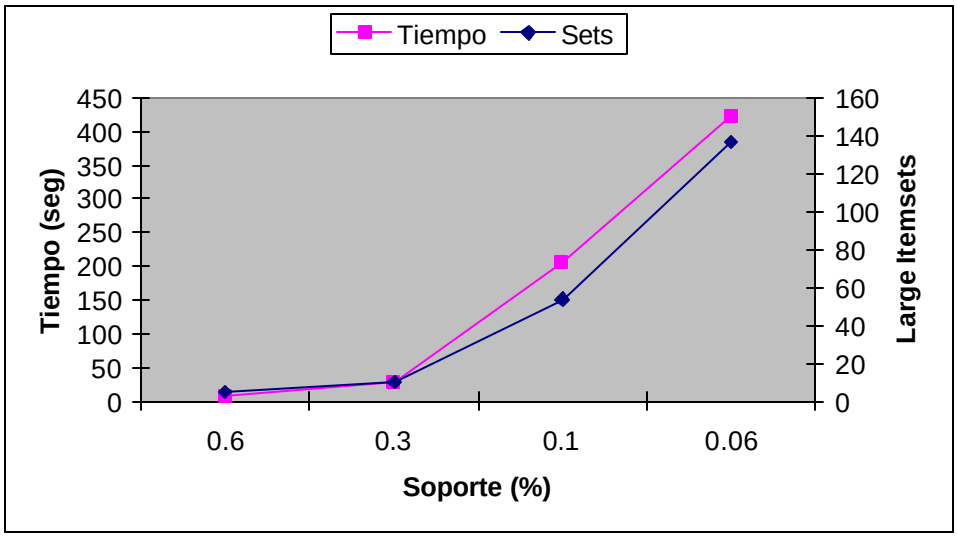
**Tabla 6.3. Resultados obtenidos para T5I3D83K con Associator**

| Soporte (%) | Tiempo (Seg.) | Número de Large Itemsets | | | |
|---|---|---|---|---|---|
| | | Total | Tam 1 | Tam 2 | Tam 3 |
| 0.6 | 10 | 30 | 25 | 5 | 0 |
| 0.3 | 10 | 73 | 63 | 10 | 0 |
| 0.1 | 11 | 476 | 422 | 41 | 13 |
| 0.06 | 10 | 1191 | 1054 | 106 | 31 |

Al comparar los resultados de las tablas 6.2 con 6.3, se observa que únicamente para el soporte mas alto (0.6%) disminuye el tiempo de ejecución. Con soporte de 0.6% resultan en promedio 30 ítems tamaño 1 que Equikeep debe evaluar por cada atributo de la tabla T5I3D83K, y con 0.06% cerca de 1000 ítems tamaño 1. Es decir, cuando el soporte mínimo disminuye, el número de itemsets generados aumenta considerablemente lo cual hace que la evaluación de la expresión lógica de la primitiva Equikeep tarde demasiado y que el tiempo de ejecución aumente de manera notoria con soportes menores. Esto no sucede con la primitiva Associator Range, en la cual los tiempos de ejecución casi se mantienen constantes con la variación del soporte mínimo.

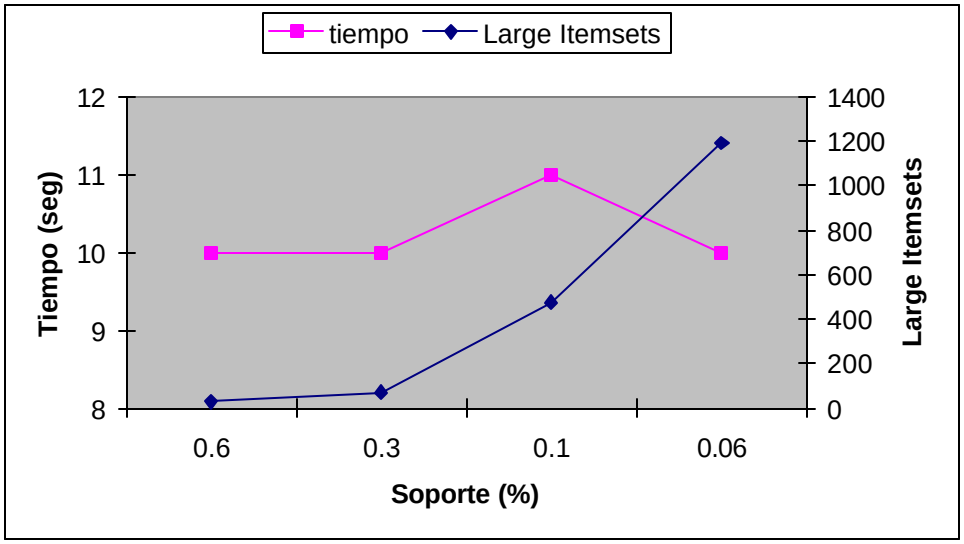
Los tiempos de ejecución presentados en las tablas 6.2 y 6.3 se resumen en las Figuras 6.1 y 6.2 respectivamente. La Figura 6.1 indica que al aplicar en conjunto las primitivas de asociación el tiempo de ejecución y el número de Itemsets tienden a aumentar linealmente cuando el soporte mínimo disminuye.

**Figura 6.1. Resultados obtenidos para T5I3D83K con Equikeep y Associator**




La Figura 6.2 indica que al aplicar únicamente la primitiva de asociación Associator Range, el número de Itemsets conserva el mismo comportamiento de la anterior prueba (tienden a aumentar linealmente cuando el soporte mínimo disminuye), sin embargo, el tiempo de ejecución casi mantiene el mismo valor para los diferentes soportes mínimos.

**Figura 6.2. Resultados obtenidos para T5I3D83K con Associator**




La tabla 6.4 muestra la confianza, el tamaño máximo y la cantidad de reglas generadas con la función Describe Association Rules, sobre los anteriores large Itemsets.

**Tabla 6.4. Resultados obtenidos para T5I3D83K con Describe Association Rules**

| Confianza (%) | Soporte Itemsets (%) | No. Reglas | Tamaño Max. Reglas |
|---|---|---|---|
| 20 | 0.6 | 8 | 2 |
| 20 | 0.3 | 13 | 2 |
| 20 | 0.1 | 57 | 3 |
| 20 | 0.06 | 112 | 3 |

En la Figura 6.3 se presentan algunas de las reglas más representativas obtenidas por medio de la función *Describe Association Rules*, con los parámetros antes mencionados, donde la primera columna representa el número de la regla generada y la segunda columna el antecedente y consecuente de la regla. Por ejemplo la regla número 113, indica que "el 70% de los clientes que compraron Fríjol Cargamento Rojo y Maizena Top también compraron Lenteja; o la regla número 58 "el 21% de los clientes que compraron Fríjol Lima y Arveja Verde también compraron Garbanzo".

**Figura 6.3. Reglas Representativas para T5I3D83K**

| Nro. Regla | Regla | |
|---|---|---|
| | Antecedentes | Consecuentes |
| 113 | 600012 ^ 600023 | 600015 |
| 58 | 600002 ^ 600017 | 600019 |

Representa:

| Nro. Regla | Regla | |
|---|---|---|
| | Antecedentes | Consecuentes |
| 113 | Fríjol Cargamanto Rojo ^ Maizena Top | Lenteja |
| 58 | Fríjol Lima ^ Arveja Verde | Garbanzo |

Para la base de datos T10I4D145K, se trabajó con soportes entre 0.5% y 0.05% y confianza del 30%. Dado que el número de itemsets frecuentes tamaño 1 esta entre 70 y 2000, la evaluación de la expresión lógica de la primitiva Equikeep tarda demasiado y aumenta significativamente el tiempo de ejecución. En la tabla 6.5 se presenta los tiempos de ejecución y el número de ítems frecuentes al aplicar en conjunto las primitivas de asociación.

**Tabla 6.5. Resultados obtenidos para T10I4D145K con Equikeep y Associator**

| Soporte (%) | Tiempo (Seg.) | Número de Large Itemsets | | | |
|---|---|---|---|---|---|
| | | Total | Tam 2 | Tam 3 | Tam 4 |
| 0.5 | 25 | 11 | 10 | 1 | 0 |
| 0.3 | 124 | 28 | 21 | 7 | 0 |
| 0.1 | 730 | 160 | 119 | 36 | 5 |
| 0.05 | 1653 | 510 | 380 | 112 | 18 |

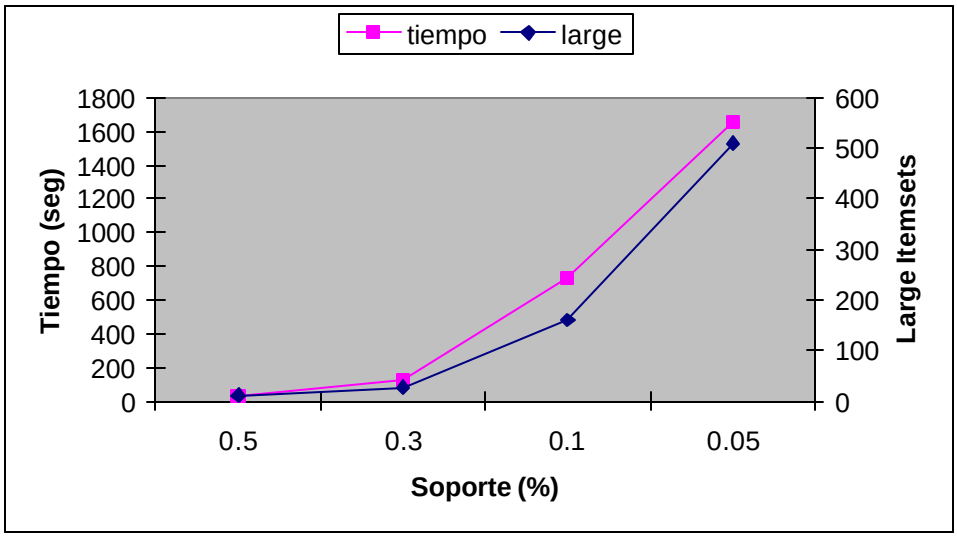
La tabla 6.6 muestra los tiempos de ejecución y el número de Large Itemsets obtenidos al aplicar únicamente la primitiva Associator Range sobre la misma base de datos.

**Tabla 6.6. Resultados obtenidos para T10I4D145K con Associator**

| Soporte (%) | Tiempo (Seg.) | Número de Large Itemsets | | | | |
|---|---|---|---|---|---|---|
| | | Total | Tam 1 | Tam 2 | Tam 3 | Tam 4 |
| 0.5 | 754 | 78 | 67 | 10 | 1 | 0 |
| 0.3 | 751 | 192 | 164 | 21 | 7 | 0 |
| 0.1 | 753 | 1049 | 889 | 119 | 36 | 5 |
| 0.05 | 751 | 2514 | 2004 | 380 | 112 | 18 |

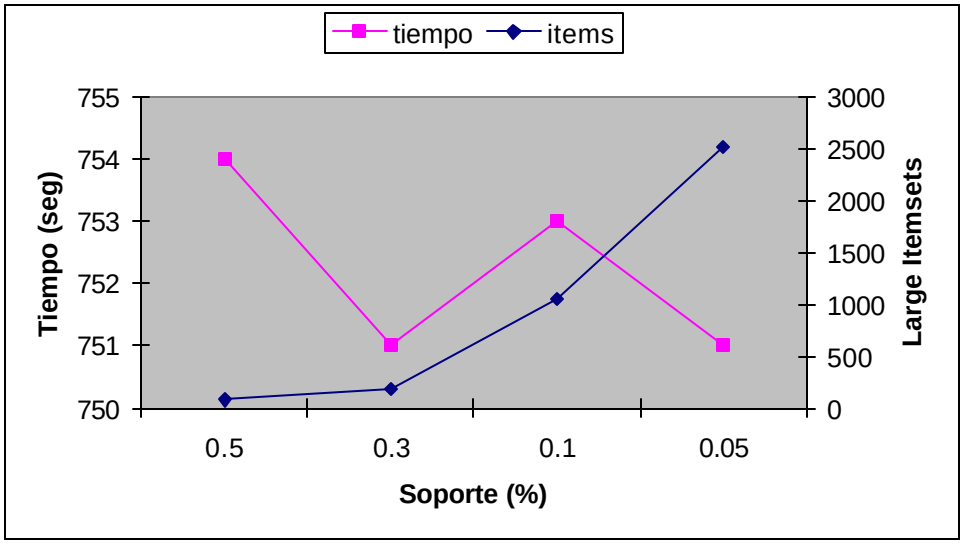
Los tiempos de ejecución presentados en las tablas 6.5 y 6.6 se resumen en las Figuras 6.4 y 6.5 respectivamente. La Figura 6.4, al igual que la Figura 6.1, indica que al aplicar en conjunto las primitivas de asociación el tiempo de ejecución y el número de Itemsets tienden a aumentar linealmente cuando el soporte mínimo disminuye.

**Figura 6.4. Resultados obtenidos para T10I4D145K con Equikeep y Associator**




La Figura 6.5 resume los tiempos de ejecución y el número total de itemsets frecuentes generados por Associator Range. Al igual que en la Figura 6.2, el número de Itemsets tiende a aumentar linealmente cuando el soporte mínimo disminuye y el tiempo de ejecución se mantiene casi constante.

**Figura 6.5. Resultados obtenidos para T10I4D145K con Associator**




La tabla 6.7 muestra la confianza, el tamaño máximo y la cantidad de reglas generadas con la función Describe Association Rules, para los large Itemsets de T10I4D145K.

**Tabla 6.7. Resultados obtenidos para T10I4D145K con Describe Association Rules**

| Confianza (%) | Soporte Itemsets (%) | No. Reglas | Tamaño Max. Reglas |
|---|---|---|---|
| 30 | 0.5 | 12 | 3 |
| 30 | 0.3 | 24 | 3 |
| 30 | 0.1 | 97 | 4 |
| 30 | 0.05 | 238 | 4 |

Conservando el formato de la Figura 6.3, se presentan algunas de las reglas más representativas obtenidas por medio del operador SQL *Describe Association Rules* en la Figura 6.6, donde la regla número 118, representa que "el 30% de los clientes que compraron Fríjol Lima y Panela también compraron Lenteja y Sal Refisal"; o la regla número 210 "el 72% de los clientes que compraron Arveja Verde, Panela y Sal Refisal también compraron Lenteja".

**Figura 6.6. Reglas Representativas para T10I4D145K**

| Nro. Regla | Regla | |
|---|---|---|
| | Antecedentes | Consecuentes |
| 118 | 600002 ∧ 600030 | 600015 ∧ 603005 |
| 210 | 600017 ∧ 600030 ∧ 603005 | 600019 |

Representa:

| Nro. Regla | Regla | |
|---|---|---|
| | Antecedentes | Consecuentes |
| 118 | Frijol Lima ∧ Panela | Lenteja ∧ Sal Refisal |
| 210 | Arveja Verde ∧ Panela ∧ Sal Refisal | Lenteja |

**6.2.3 Análisis de Resultados.** El tiempo de ejecución para el cálculo de los itemsets frecuentes de una base de datos, crece linealmente con el número de los registros (transacciones), pero exponencialmente con el grado (número de atributos) de la tabla. Sin embargo, además del número de itemsets frecuentes y de los tamaños de los itemsets, el valor del soporte es un factor importante que también determina el tiempo de ejecución, porque el número de ítems frecuentes al igual que el número de reglas generadas aumenta cuando el valor de éste parámetro disminuye.

Al comparar los tiempos de ejecución de las primitivas Associator y Equikeep para T5I3D83K (tabla 6.2) con los tiempos de los operadores relacionales implementados en un

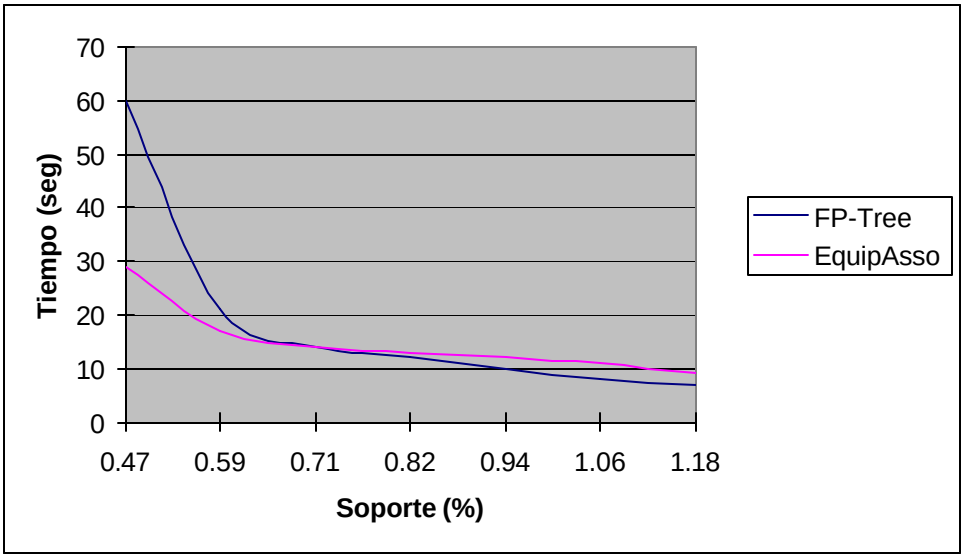
sistema con una arquitectura débilmente acoplada [Calderon et al] sobre el mismo conjunto de datos (tabla 6.8), se comprobó los beneficios de tener una herramientas con estas características, ya que reduce notablemente los tiempos de ejecución y permite de una manera versátil, eficiente y escalable, algunas tareas básicas de descubrimiento de conocimiento, sacando provecho y potencia del SGBD PostgreSQL.

**Tabla 6.8. Resultados obtenidos para T5I3D83K Proyecto Taryi**

| | Tiempo(sg) | | Número de Itemsets | |
|---|---|---|---|---|
| **Soporte (%)** | **FP-Tree** | **EquipAsso** | **Tipo 2** | **Tipo 3** |
| 0.47 | 59.59 | 29.15 | 59 | 15 |
| 0.59 | 21.32 | 17.07 | 37 | 8 |
| 0.71 | 13.68 | 13.69 | 23 | 6 |
| 0.82 | 11.78 | 12.53 | 19 | 3 |
| 0.94 | 9.77 | 12.3 | 16 | 2 |
| 1.06 | 8.29 | 11.64 | 15 | 1 |
| 1.18 | 7.51 | 8.67 | 12 | 0 |

Los resultados obtenidos para esta prueba se resumen en la figura 6.7.

**Figura 6.7. Resultados obtenidos para T5I3D83K Proyecto Taryi**



De acuerdo al análisis total de las reglas válidas obtenidas de las 2 muestras (190 reglas para T5I3D83K, 371 reglas para T10I4D145K), la gran mayoría incluye 8 productos básicos (Fríjol Lima, Lenteja, Arveja Verde, Garbanzo, Fríjol Cargamento Rojo, Maizena Top, Sal Refisal, Panela), el resto se distribuye uniformemente entre los demás productos. Esto indica la preferencia de compra de los clientes por los productos de primer orden de la canasta familiar, a pesar de la gran cantidad de artículos (10.757) registrados en las bases de datos.

## 6.3 RESULTADOS OBTENIDOS PARA CLASIFICACION

El proceso de prueba del algoritmo se ha realizado con el objetivo principal de comparar sus tiempos de ejecución con respecto al volumen de registros. Las pruebas se realizaron sobre dos conjunto de datos con un número pequeño de registros, para verificar la confiabilidad de la respuesta y luego se duplica hasta obtener la cantidad deseada, con el fin de conocer el comportamiento de la primitiva Mate by con respecto al tiempo.

**6.3.1 Pruebas con Conjuntos de Datos Reales.** Para realizar las pruebas de los algoritmos con datos reales, se seleccionaron los conjuntos de datos de Zoológico y de Autos obtenidos del repositorio de datos de Machine Learning (uci).

**6.3.1.1 Base de datos de Zoológico.** Esta base de datos contiene la información de 101 animales, con 18 atributos (nombre del animal, 15 atributos booleanos y 2 atributos numéricos).

**Características de la Base de datos**

- Clases con su respectivo número de animales

  - Clase 1 (41 animales)
  - Clase 2 (20 animales)
  - Clase 3 (5 animales)
  - Clase 4 (13 animales)
  - Clase 5 (4 animales)
  - Clase 6 (8 animales)
  - Clase 7 (10 animales)

- Número de valores con atributos faltantes: Ninguno
- Información de los atributos:

**Tabla 6.9. Características base de datos zoo.data**

| Nro | Nombre del atributo | Tipos de valores del dominio |
|---|---|---|
| 1 | Nombre del animal | Único para cada instancia |
| 2 | Pelo | Booleano |
| 3 | Plumas | Booleano |
| 4 | Huevos | Booleano |
| 5 | Leche | Booleano |
| 6 | Volador | Booleano |
| 7 | Acuático | Booleano |
| 8 | Cazador | Booleano |
| 9 | Dientes | Booleano |
| 10 | Esqueleto | Booleano |
| 11 | Respira | Booleano |
| 12 | Venenoso | Booleano |
| 13 | Aletas | Conjunto de valores {0,2,4,5,6,8} |
| 14 | Patas | Booleano |
| 15 | Cola | Booleano |
| 16 | Doméstico | Booleano |
| 17 | Tamaño_Gato | Booleano |
| 18 | Class | Valores enteros en el intervalo [1..7] |

**6.3.1.2 Base de datos de Autos.** Esta base da datos contiene la información de 392 carros, con 8 atributos (6 numéricos y 2 categóricos).

**Características de la base de datos**

**Tabla 6.10. Características base de datos auto-mpg.data**

| Nro | Atributo | Valores del Dominio |
|---|---|---|
| 1 | MPG | Continuo |
| 2 | Cylinders | Continuo |
| 3 | CubicInches | Continuo |
| 4 | Horsepower | Continuo |
| 5 | WeightLbs | Continuo |
| 6 | Year | 1983, 1981, 1978, 1982, 1973, 1977, 1974, 1975, 1979, 1976, 1980, 1971, 1972. |
| 7 | Brand | datsun, subaru, onda, mazda, toyota, ford, buick, capri, mercury, amc, |

| | | |
|---|---|---|
| | | plymouth, chevrolet, dodge, chrysler, pontiac, peugeot, vw, opel, renault, audi, mercedes, oldsmobile, volvo, bmw, fiat, cadillac, nissan, saab, hi, triumph. |
| 8 | Class | US, Japan, Europe. |

**6.3.2 Pruebas de Rendimiento.** Este conjunto de pruebas se realizaron para observar el comportamiento de la primitiva Mate by cuando el número de registros aumenta; el conjunto de datos utilizado para las pruebas fue zoo.data y auto-mpg.data, los cuales se duplican hasta obtener el número de registros que se requiere.

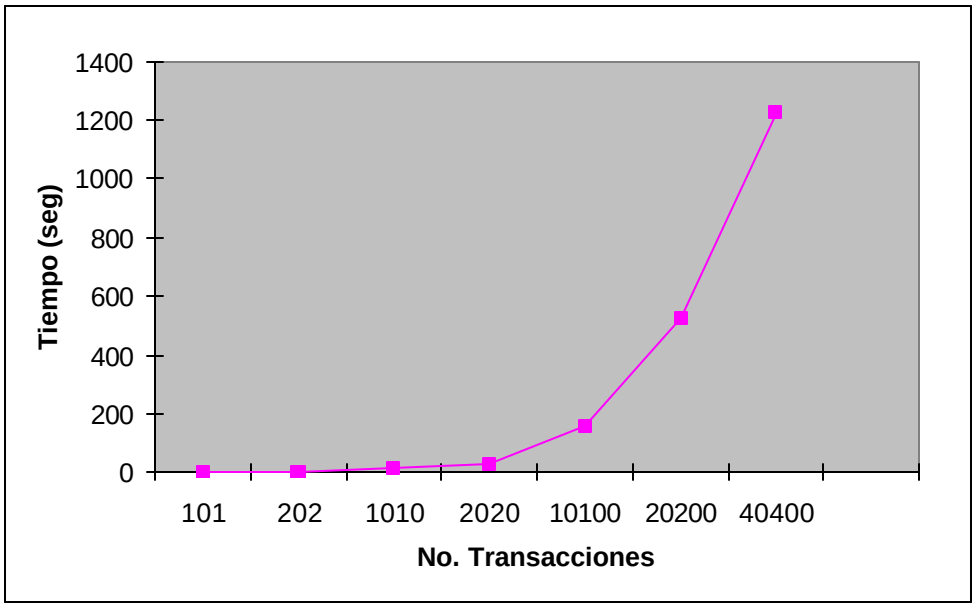
El tiempo expresado en la Tabla 6.11 y 6.12 está dado en segundos.

**6.3.2.1 Prueba 1.** Esta prueba corresponde al repositorio zoo.data y los resultados son los siguientes:

**Tabla 6.11. Resultados zoo.data**

| **Prueba 1** | | |
|---|---|---|
| **No. Transacciones** | **Tiempo (seg)** | **Resultado Mate** |
| 101 | 0.8 | 6372 |
| 202 | 2.6 | 6372 |
| 1010 | 13.3 | 6372 |
| 2020 | 28.2 | 6372 |
| 10100 | 156.4 | 6372 |
| 20200 | 528.5 | 6372 |
| 40400 | 1229.2 | 6372 |

**Figura 6.8. Resultados obtenidos al variar número de transacciones para zoo.data**

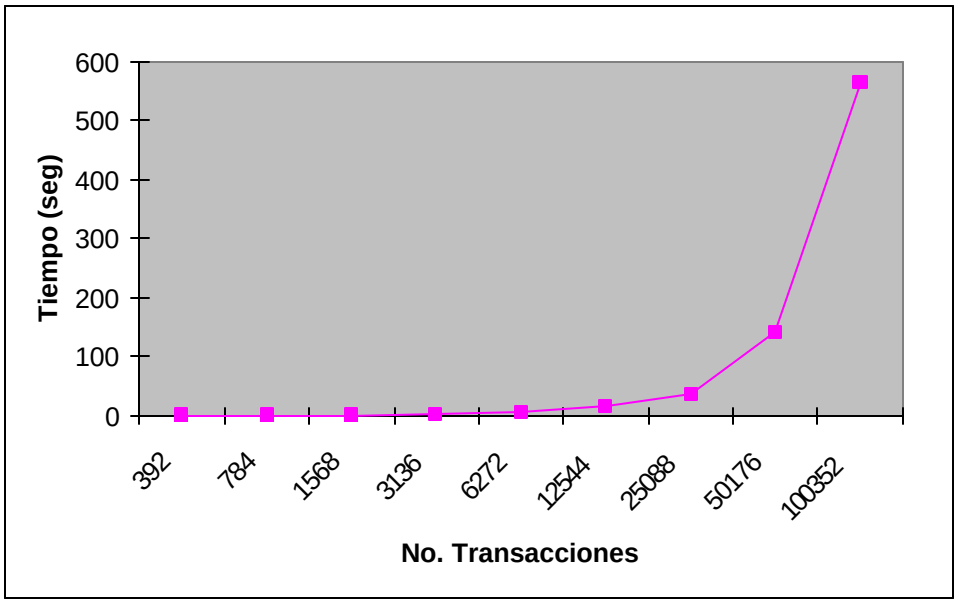



**6.3.2.2 Prueba 2.** Esta prueba corresponde al repositorio auto-mpg.data y los resultados son los siguientes:

**Tabla 6.12. Resultados auto-mpg.data**

| Prueba 1 | | |
|---|---|---|
| **No. Transacciones** | **Tiempo (seg)** | **Resultado Mate** |
| 392 | 0.7 | 4605 |
| 784 | 0.8 | 4605 |
| 1568 | 1.6 | 4605 |
| 3136 | 3.2 | 4605 |
| 6272 | 7.3 | 4605 |
| 12544 | 17.7 | 4605 |
| 25088 | 36.4 | 4605 |
| 50176 | 140.3 | 4605 |
| 100352 | 564 | 4605 |

**Figura 6.9. Resultados obtenidos al variar número de transacciones para auto-mpg.data**




**6.3.2.3 Pruebas con entro( ) y gain( ).** Como se planteó en el capítulo 5, la ejecución de estas funciones requiere el procesamiento previo de Mate by. El hecho que durante las pruebas de Mate by se descubriera que cuando el número de intervalos de discretización se mantiene constante para todos y cada uno de los atributos aún cuando el número de registros aumente significativamente, el resultado que produce Mate by, es decir el número de combinaciones válidas posibles se mantiene también constante, permite hacer la siguiente afirmación:

Puesto que la entrada para operar de entro(), es el resultado de Mate By, entradas iguales en número de atributos, número de registros y combinaciones, consumirán siempre la misma cantidad de recursos en tiempo y espacio, despreciando el hecho que los valores sobre los que se calcula la entropía son mayores en valor numérico, pero conservaran las proporciones.

De igual manera si entro() proporciona salidas constantes, que se utilizan como entrada para gain(), se concluye que el consumo de recursos de gain() no tiene cambios significativos para casos como el mencionado.

Naturalmente, para casos donde las tablas presenten un crecimiento en grado, dado el mayor número de combinaciones posibles, entro() consumirá mas recursos, reflejado más en tiempo que en espacio.

Para el caso de gain(), puesto que el árbol que se construye crece en profundidad a razón de nivel por atributo adicional, es razonable afirmar que alcanzará un máximo de niveles como

atributos contenga la tabla. El crecimiento del árbol en anchura, en cambio, esta relacionado con el número de intervalos de discretización que tiene cada atributo, es decir, que un nodo que representa un atributo tendrá tantos hijos como valores posibles pueda tomar ese atributo. En este tipo de casos, con atributos que puedan tomar muchos valores, el consumo de recursos, los tiempos de recorridos, las búsquedas, operaciones que se hagan sobre el árbol y estructuras adicionales se incrementarán notoriamente.

La fase de generación de las reglas depende directamente del tamaño del árbol que se haya generado, pues el número de recorridos que se hagan dependerá del número de nodos, por esto, el consumo de recursos en la creación de la tabla 'tclases' que guarda todos los nodos se refleja más en espacio que en tiempo. En cambio, los recursos consumidos en crear 'trulesclases' se inclinan en mayor cantidad de tiempo debido a los recorridos selectivos para árboles grandes, pero el espacio de almacenamiento depende de la cantidad de reglas generadas.

La fase final adicional que es utilizar la información contenida en 'mate_discret' para obtener la información real de los valores discretizados en las reglas obtenidas, conlleva un bajo consumo de tiempo y más aún en espacio. Sin embargo, dependerá de la cantidad de reglas generadas, y sobre todo de los atributos presentes y los intervalos de discretización.

**6.3.3 Análisis de Resultados.** Se puede concluir que a mayor número de intervalos de discretización, mayor es el número de ítems generados por Mate by debido a que aumenta la dispersión de los atributos condición con respecto a la clase de decisión

Los resultado generales al aplicar la primitiva Mate by, muestran que los tiempos de ejecución están directamente relacionados con el número de atributos y las distintas categorías del atributo clase, e inversamente proporcional al número de registros a procesar.

Todos los registros que se agregaron a las bases de datos, con el fin de aumentar la cardinalidad de la tabla hasta aprovechar la máxima capacidad de procesamiento de la máquina, pertenecen a las mismas divisiones de clases, así el número de ítems clase obtenidos con el primer conjunto de transacciones, es el mismo que se obtiene al procesar el último conjunto, es decir que no hay aumento en el número de categorías clase.

Si se introdujeran datos nuevos que contengan valores de dominio no utilizados en las primeras evaluaciones, se generarían entonces nuevas categorías de clasificación.

La primitiva Mate by fue sometida a un conjunto de pruebas cuyos resultados garantizan la eficiencia del mismo así como la calidad de los resultados obtenidos mientras cumpla con una serie de requerimientos.

Las pruebas de desempeño de la primitiva demostraron que:

- El tiempo de ejecución de la primitiva sobre conjuntos de datos pequeños es muy bajo. La diferencia en tiempo de ejecución se hace más notoria en la medida en que el conjunto de datos es mayor, es decir, el tiempo de ejecución de la primitiva crece linealmente con respecto al tamaño del conjunto de datos.

- El tiempo de ejecución de la primitiva crece significativamente de acuerdo al grado de la tabla aunque el tamaño del conjunto de datos sea menor, es decir, la eficiencia de procesamiento de la primitiva depende de forma inversamente proporcional a la cardinalidad y el grado de la tabla.

- La complejidad de la primitiva Mate by esta directamente relacionada con el número de valores distintos de los atributos categóricos y la cantidad de registros en el conjunto de datos.

# 7. CONCLUSIONES

En este proyecto se implementaron al interior del SGBD PostgreSQL, los nuevos operadores algebraicos y primitivas SQL para el descubrimiento de reglas de Asociación y Clasificación propuestas por Timarán [Tima02b]. Producto de todo el trabajo realizado se concluye que es necesario seguir una metodología que permita adicionar nuevas capacidades al SGBD PostgreSQL, sistemáticamente y conservando la arquitectura interna y estructura de este SGBD así:

- Realizar un estudio funcional y estructural detallado de cada una de las etapas del procesador de consultas de PostgreSQL (parser, planner/optimizer, executor) que permita identificar la estructura de un operador Postgres y los nodos básicos que lo conforman.

- Desarrollar los procedimientos respectivos en el lenguaje anfitrión C, de acuerdo a la arquitectura de los operadores de postgres, en cada una de las etapas del procesador de consultas, creando nuevos tipos, nodos y/o rutinas de interfaz donde sea necesario.

- Utilizar la capacidad de Postgres de definición de funciones definidas por el usuario (FDU) a través de lenguajes procedurales, aprovechando las ventajas de programación en servidor cuando no se requiera la implementación de procedimientos que modifiquen las etapas del procesador de consultas.

Se cuenta con una versión de PostgreSQL con la capacidad de extraer reglas de asociación y clasificación en una arquitectura fuertemente acoplada.

Analizando las pruebas obtenidas para Asociación, en esta arquitectura fuertemente acoplada con Postgres con respecto a una débilmente acoplada (proyecto Taryi [Calderon et al]), el rendimiento mejora significativamente en la primera. Los resultados demuestran la eficiencia de una arquitectura fuertemente acoplada y lo complejo de su implementación.

El tiempo de ejecución para los nuevos operadores no disminuye, se mantiene similar cuando se cambia el modo de acceso secuencial (Seqscan) de la tabla por accesos controlados por índices (Indexscan), debido a que se accede solo una vez a los registros de la tabla, y por cada tupla que entregue el nodo inferior (Seqscan o Indexscan), se generan las diferentes asociaciones.

Los nuevos operadores de Asociación y Clasificación son mas eficientes en cuanto a tiempo de ejecución cuando se emplean técnicas de discretización de forma global (antes de comenzar la ejecución de los operadores), pues el uso de datos sin discretizar se traduce generalmente en mayor consumo de recursos computacionales.

Al ejecutar una misma consulta repetidas veces, los tiempos de respuesta van disminuyendo debido a las características del optimizador de Postgres, ya que no repite los cálculos de estimación de costos de los diferentes paths del plan de ejecución, sino que utiliza un criterio certero para los histogramas, que le permite obtener los datos con mas rapidez porque están almacenados en lugares contiguos.

Los tiempos de ejecución de las reglas de asociación y clasificación son despreciables con respecto a los tiempos de ejecución de la generación de los itemsets frecuentes y la construcción del árbol de decisión.

A partir de los resultados obtenidos de las pruebas para Asociación se concluye que:

- El tiempo de ejecución de los operadores de asociación crece linealmente con respecto al número de transacciones de la base de datos y exponencialmente con el número de atributos de la relación.

- El valor del soporte es un factor importante que también determina el tiempo de ejecución, porque el número de ítems frecuentes al igual que el número de reglas generadas aumenta cuando el valor de éste parámetro disminuye.

- La primitiva Equikeep hace más eficiente la generación de itemsets frecuentes cuando se trabaja con altos valores de soporte y disminuye en eficiencia cuando los valores de soporte son menores debido a que aumentan el número de ítems tamaño uno (1) que hacen parte de las expresiones lógicas que se deben evaluar.

A partir de los resultados obtenidos de las pruebas para Clasificación se concluye que:

- Cuando aumenta el número de valores posibles de un atributo, también aumenta el número de ítems generados por Mate by debido a que es mayor la dispersión de los atributos condición con respecto a la clase de decisión.

- Los resultado generales al aplicar la primitiva Mate by, muestran que los tiempos de ejecución están directamente relacionados con el número de atributos y las distintas categorías del atributo clase, e inversamente proporcional al número de registros a procesar.

Una vez que se han descrito los resultados más relevantes que se han obtenido durante la realización de este trabajo, se sugiere una serie de puntos de partida para futuros trabajos:

- Realizar mayores pruebas de rendimiento de esta arquitectura e implementar otras primitivas que Timarán [Tima02b] ha propuesto para Asociación y Clasificación.

- Mejorar aún más el rendimiento de esta arquitectura implementando las estrategias de transformación y costos que Timarán propone para los operadores algebraicos relacionales que soportan estas primitivas.

- Implementar las funciones definidas por el usuario al interior del SGBD PostgreSQL como nuevos operadores.

- Implementar otros operadores algebraicos y primitivas SQL para otras tares de minería de datos y de todo el proceso de KDD.

- Desarrollar una interfaz gráfica para minar todo tipo de reglas.

- Probar el modelo de clasificación con datos reales

- Implementar un nuevo operador que aplique el modelo de clasificación construido y clasifique datos cuya clase se desconoce.

- Liberar una versión de PostgreSQL con la capacidad de descubrir conocimiento en bases de datos.

Finalmente, este trabajo permitió aplicar los conocimientos adquiridos en el programa de Ingeniería de Sistemas y sobretodo los de la electiva Base de Datos III, en la cual se abordó temas relacionados a las técnicas de Minería de datos y del descubrimiento de conocimiento en base de datos (KDD).

## BIBLIOGRAFIA


[Agrawal et.al 92] AGRAWAL, R., GHOSH, S., IMIELINSKI, T., IYER, B., SWAMI, A. An Interval Classifier for Database Mining Applications, Proc. of the 18th VLDB Conference, Vancouver, British Columbia, Canada, 1992.

[AS94] AGRAWAL, R AND SRIKANT, R. Fast algorithms for mining association rules. VLDB-94, 1994.

[Calderon et al.] CALDERON, A., RAMIREZ, I., GUEVARA, F., ALVARADO, J, T, R., Proyecto Taryi., Sistema de Investigaciones Udenar, San Juan de Pasto, 2005.

[Clear et al. 99a] 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.

[FONG86] FONG, Z. The Design and implementation of the Postgres Query Optimizer; University of California, Berkeley, Computer Science Department, Agosto 1986.

[HS93] HOUTSMA, M., SWAMI. A. Set-oriented mining of association rules. Research report RJ 9567, IBM Almaden research center, San Jose, California. Octubre de 1993.

[LOCK98a] LOCKHART, T. The PostgreSQL Programmer's Guide, The PostgreSQL Global Development Group.

[MOM97] MOMJIAN, B. PostgreSQL Backend Directories; código fuente: src/tool/backend/backend_dirs.html; Diciembre 1997.

[MOM98] MOMJIAN, B. How PostgreSQL Processes a Query; código fuente: src/tool/backend/index.html; Agosto 1998.

[MOS03] MOSSOS, N. Estudio y desarrollo de una aplicación de visualización del proceso de optimización lógica de una consulta sql en el sistema manejador de bases de datos postgresql; Cali, 2003.

[OJ01] ORTIZ, J. Implementación de un operador de clasificación basado en la técnica de árboles de decisión para el manejador de bases de datos de Postgres. Universidad del Valle, 2001.

[SIMK98] SIMKOVICS, S. Enhancement of the ANSI SQL Implementation of PostgreSQL, 1998.

[STRW86] STONEBRAKER, M., ROWE, L. The design of Postgres; ACMSIGMOD ’86 Conference on Management of Data, Mayo 1986.

[Tima01] 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.

[Tima02a] TIMARÁN, R. Descubrimiento de conocimiento en Bases de Datos: Una visión general, en memorias del Primer Congreso Nacional de Investigación y Tecnología en Ingeniería de Sistemas, Universidad del Quindío, Armenia, octubre de 2002.

[Tima02b] TIMARÁN, R. Nuevos operadores algebraicos y Primitivas SQL para el Descubrimiento de Conocimiento en Bases de Datos, informe avance tesis doctoral, Universidad del Valle, diciembre 2002.

[TiMM03] TIMARÁN, R., MILLÁN, M., MACHUCA, F. 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, mayo 2003.

# ANEXO A

## ANÁLISIS DE LA ESTRUCTURA Y CÓDIGO FUENTE DE EL MANEJADOR DE BASES DE DATOS POSTGRESQL

Teniendo en cuenta el conocimiento de la arquitectura y modo de funcionamiento del SGBD PostgreSQL, se realiza un análisis detallado de su estructura y código fuente para poder abordar a cabalidad lo propuesto en el presente proyecto.

Es necesario realizar este análisis ya que se busca implementar operadores y funciones agregadas propias de minería de datos de una manera fuertemente acoplada con el gestor.

Dentro de los archivos estudiados se nombran librerías que también se han analizado. Esta información se puede encontrar en el Anexo B con más detalle al final del documento.

**ANALISIS PARSER**

Tras un análisis de los códigos fuente del Parser de PostgreSQL, podemos deducir las siguientes ideas o conceptos:

Teniendo en cuenta que existen librerías propias para cada proceso de PostgreSQL (i.e. Parser, Rewrite, Planner, Optimizer, etc.) se puede dar cuenta que muchas de estas librerías son compartidas y que en su gran mayoría presentan la creación de estructuras y/o funciones para el funcionamiento propio de Postgres. Cuando se dice estructuras se menciona variables de entorno, algunas variables privadas dependiendo de la función o el archivo que las llame, estructuras de información como: árboles, listas y los mismos nodos que serán utilizados de una o varias maneras en el funcionamiento del gestor y funciones de creación de estructuras, llamadas a sistema o las mismas funciones de operación y manejo de información o datos de Postgres.

También se puede observar que en el código fuente de PostgreSQL se encuentra el paradigma de la programación estructurada ya que existen dentro de él, funciones y estructuras con tales características.

Entrando ya en el tema del código fuente se pueden diferenciar los siguientes archivos, entre otros, dentro del directorio que hace parte del Parser:

| | |
|---|---|
| parser.c | Las tareas de parser.c comienzan aquí. |
| scan.l | Descompone la consulta en Tokens. |
| scansup.c | Manejo de Caracteres de escape. |
| keywords.c | Convierte las palabras reservadas en el token específico. |
| gram.y | Analiza los tokens y llena la estructura ParserTree. |
| analyze.c | Manejo del proceso Post-Parse para cada tipo de consulta. |
| parse_clause.c | Manejo de las cláusulas como where, order by, group by, etc. |
| parse_coerce.c | Usado para el cast de expresiones de diferente tipo. |
| parse_expr.c | Manejo de expresiones como col, col + 3, x = 3 or x = 4. |
| parse_oper.c | Maneja las operaciones de las expresiones. |
| parse_agg.c | Manejo de funciones agregadas como sum(col1), avg(col2). |
| parse_func.c | Manejo de funciones. |
| parse_node.c | Crea los nodos de las estructuras. |
| parse_target.c | Maneja la Lista resultante de la consulta |
| parse_relation.c | Soporta rutinas para el manejo de tablas y columnas. |
| parse_type.c | Soporta rutinas para el manejo de tipos. |

**Archivo GRAM.Y**

Encargado de las reglas/acciones YACC en POSTGRESQL. En general, nada en este archivo debe iniciar accesos a la base de datos ya que dicho código fallará cuando se aborte la actual transacción. Cualquier elemento que dependa de la base de datos o del cambio de estados, debe ser manejado dentro de parse_analyze().

Gram.y esta estructurado en tres partes: las definiciones, las reglas y las subrutinas. En las definiciones llama a las librerías y crea las estructuras necesarias. En la parte de las reglas, esta definida la gramática de PostgreSQL y es en donde se modificó parte del código. La rutina void parser_init(void) hace parte de la última sección.

**Archivo analyze.c**

Transforma el parse tree al query tree.

Incluye dentro de su código las siguientes librerías (entre otras):

parser/analyze.h
parser/gramparse.h
parser/parsetree.h
parser/parse_agg.h
parser/parse_clause.h
parser/parse_coerce.h
parser/parse_oper.h
parser/parse_relation.h
parser/parse_target.h

Entre sus funciones se encuentran:

- **parse_analyze**

Analiza un parse tree natural (crudo) y lo transforma a la forma de Query. El resultado es una lista de nodos query. Las declaraciones optimizables requieren de una transformación considerable.

- **TransformStmt**

Transforma un Parse tree a un Query tree.

- **makeObjectName()**

Crea un nombre para un índice, secuencia, coacción, etc.

- **TransformIndexStmt**

Transforma la cualificación de la declaración de los índices.

- **TransformRuleStmt**

Transforma un Rule Statement creado. Las acciones son una lista de parse trees los que son transformados a una lista de query trees.

- **TransformSelectStmt**

Transforma el Select Statement, hace disponible la cláusula For Update, procesa la cláusula From, transforma el targetList y el Where, también inicializa el proceso del Having como una cláusula Where.

- **transformSetOperationsStmt y transformSetOperationTree**

Aquí se presenta parte de la transformación de un SelectStmt a un Query.

- **GetSetColTypes**

Saca los tipos de columna de un nodo ya transformado.

- **ApplyColumnNames**

Adhiere nombres de columnas desde la una lista ColumnDef a la lista TargetEntry.

- **RelationHasPrimaryKey**

Mira si una relación tiene una llave primaria.

- **TransformConstraintAttrs**

Procesa la lista de columnas presentes en las cláusulas de coacción y detecta inconsistencias en los mismos.

- **analyzeCreateSchemaStmt**

Analiza la declaración "create schema". Separa los elementos de la lista de esquema dentro de comandos individuales y los coloca en la lista de resultados. Esta lista de resultados es una lista de nodos del parser que aun necesitan ser analizados. Es llamada por commands/command.c.

- **transformPrepareStmt**

Transforma la lista TypeNames a la lista y arreglo de type OID.

También está entre sus funciones transformar las declaraciones de inserción y eliminación, da un manejo especial de los tipos definidos para una columna, compila los nodos FromExpr, hace los chequeos para querys "Select for update", transforma los UpdateStmt y AlterTableStmt, transforma las definiciones de las columnas y transforma la declaración de Create Table.

**Archivo parse_clause.c**

Maneja las cláusulas "ORDER BY", "GROUP BY", "DISTINCT ON" en el parser

Incluye dentro de su código las siguientes librerías (entre otras):

postgres.h
parser/analyze.h
parser/parsetree.h
parser/parse_coerce.h
parser/parse_oper.h
parser/parse_relation.h
parser/parse_target.h

Entre sus funciones se encuentran:

- **transformFromClause**

Procesa la cláusula FROM y adhiere ítems a los query's range table, joinlist, y namespace.
Utiliza las listas del parse_state (creadas en el parse_node.h) p_rtable, p_joinlist, y p_namespace

- **setTargetTable**

Adiciona al range table la relación destino (target table) de INSERT/UPDATE/DELETE, y crea los links especiales para ello en el ParseState.
Para el INSERT, no se necesita que el destino sea parte de un join, esto es una destinación de tuplas, no una fuente.
Para el UPDATE/DELETE, no se necesita escanear o hacerle un join al destino.
Se retorna el índice del rangetable de las relaciones destino.

- **transformJoinUsingClause (Natural Join)**

Construye una cláusula ON completa desde una lista USING parcialmente transformada.
Se dan las listas de los nodos representando las columnas izquierda y derecha relacionadas.
El resultado es una expresión cualificativa transformada

- **transformFromClauseItem**

Transforma un ítem de la cláusula FROM, adhiriendo cualquier entrada requerida al range table list, y retorna el ítem transformado listo para incluirse en el joinlist y el namespace.

- **buildMergedJoinVar**

Genera una conveniente expresión de reemplazo para una columna de MergeJoin

- **findTargetlistEntry**

Retorna el targetlist entry relacionando nodo (no transformado) dado. Si no existen entradas relacionadas, una es creada y adicionada al target list como un nodo "resjunk".

- **addAllTargetsToSortList**

Se asegura que todos los destinos (targets) en el targetlist están en la lista del ORDER BY adicionando los no-ordenados-todavía al final de la lista. Esto se usa típicamente para ayudar a implementar el SELECT DISTINCT. Retorna la lista ORDER BY actualizada.

- **addTargetToSortList**

Si los targetlist entry dados no están en la lista ORDER BY aún, los adiciona al final de la lista, usando el sortop con los nombres dados o cualquier operador de ordenamiento disponible. Retorna la lista ORDER BY actualizada.

- **targetIsInSortList**

Pregunta si los target ítem ya están en la sortlist; funciona para las listas SortCluse y GruopClause. La razón principal para necesitar esta rutina es que un TargetListEntry puede estar en una sola de las listas.

También está entre sus funciones transformar una RangeVar (referencia de relación simple), transformar un sub-SELECT que aparece en el FROM, transformar una llamada a función que aparece en el FROM, transformar las cláusulas GROUP BY, SORT, DISTINCT o DISTINCT ON, para esta última puede ser necesario adherir ítems a la lista de la cláusula Sort del query.

**Archivo parse_coerce.c**

Maneja los tipos de coerciones/comversiones para el parser.

Incluye dentro de su código las siguientes librerías (entre otras):

postgres.h
parser/parse_coerce.h
parser/parse_expr.h
parser/parse_func.h
parser/parse_type.h

Entre sus funciones se encuentran:

- **coerce_to_target_type**

Convierte una expresión a un target type y typmod.
Este es un punto de entrada de propósito general para operaciones de coerción de tipo arbitrario. El uso directo de los componentes de operación can_coerce_type, coerce_type, y coerce_type_typmod puede ser restringido para casos especiales. Esta retorna el posible árbol de expresión transformado o NULL si el tipo de conversión no es posible.

- **coerce_type**

Convierte una expresión a un tipo diferente.
El llamado ya tendría determinado que coerción es posible; mira la función can_coerce_type.

- **can_coerce_type**

Pregunta si los input_typeids pueden coerced a target_typeids.

- **coerce_type_constraints**

Crea un árbol de expresión para reforzar las constantes que serian aplicadas por el tipo. Esto solo le interesaría a los tipos de dominio.
Debe anotarse que el árbol resultado no garantiza mostrar el correcto exprType() para el dominio, este puede mostrar el tipo base.

- **coerce_type_typmod**

Forza un valor a un typmod particular, si es necesario y posible; esto es aplicado a los valores que van a ser presentados en una relación.
El llamado debe haberse asegurado que el valor es del tipo correcto, típicamente para aplicar coerce_type.
Debe anotarse que este no es necesario para trabajar en tipos de dominio, porque cualquier coerción de typmod's para un dominio es considerada parte del tipo de coerción necesitada para producir el valor del dominio en primer lugar, entonces no consigue un BaseType.

- **coerce_to_boolean**

Coacciona un argumento de un constructor que requiere entradas booleanas (AND, OR, NOT, etc), también revisando que la entrada no sea una colección.

- **select_common_type**

Determina el supertipo común de una lista de tipos de expresión de entrada. Esta es usada para determinar el tipo de salida de las construcciones de CASE y UNION.

- **coerce_to_common_type**

Coacciona una expresión al tipo conseguido. Esta es usada siguiendo select_common_type para coaccionar las expresiones individuales a un desired type. "context" es una frase usada en el mensaje de error si la coacción falla.

- **TypeCategory**

Asigna una categoría a un OID especificado.

- **IsPreferredType**

Chequea si el tipo es un tipo preferente.

- **PreferredType**

Retorna el OID de tipo preferente para la categoría especificada

- **IsBinaryCoercible**

Chequea si la scrtype es coercible a targettype

**Archivo parser_expr.c**

Encargado de manejar las expresiones dadas en las cláusulas de la consulta.

Entre sus funciones se encuentran:

- **Node *transformExpr()**

Analiza y transforma las expresiones. El optimizador y el ejecutor no pueden manejar las expresiones recogidas por el parser tree. Por lo tanto la transformación se hace aquí.

**Archivo parser_oper.c**

Encargado de manejar las operaciones dadas en las expresiones presentes en los nodos A_expr.

Entre sus funciones se encuentran:

- **Oid LookupOperName**

Determina la posición del operador y si son del mismo tipo los operandos.

- **Oid LookupOperNameTypeNames**

Busca el operador indicado, en caso de no poder encontrarlo indica el error.

- **Oid any_ordering_op**

Selecciona el orden para el operador indicado.

- **static Oid binary_oper_exact**

Reviza la igualdad de los tipos de los operandos en caso de operadores binarios.

- **Operator oper**

Busca un operador binario dado el nombre del operador y el tipo de los dos operadores.

- **Operator compatible_oper**

Busca un operador compatible, indicando que operación se desea realizar y el tipo de los dos operadores.

- **Operador right_oper**

Busca un operador adecuado indicando el tipo de dato manejado por derecha.

- **Operador left_oper**

Busca un operador adecuado indicando el tipo de dato manejado por izquierda.

- **static void op_error**

Da el mensaje de error cuando el tipo de operador no se encuentra para los dos operadores.

- **static void unary_op_error**

Da el mensaje de error cuando el tipo de operador no se encuentra para alguno de los dos operadores.

**Archivo parse_agg.c**

Maneja los agregados en el parser.

Incluye dentro de su código las siguientes librerías (entre otras):

postgres.h
parser/parse_agg.h
parser/parsetree.h

Entre sus funciones se encuentran:

- **check_ungrouped_columns**

Revisa el árbol de la expresión dada buscando variables no agrupadas (variables que no han sido listadas en las listas de groupClauses y no tienen argumentos de funciones agregadas). Emite un conveniente mensaje de error si alguno es encontrado (Se asume que la cláusula dada ha sido transformada convenientemente en la salida del parser).

- **parseCheckAggregates**

Chequea por agregados donde ellos no deberían estar y por agrupamiento inapropiado.
Idealmente esto sería hecho antes, pero es difícil distinguir los agregados desde funciones sencillas al nivel gramatical, en lugar de ello se chequea aquí. Esta función sería llamada después de que el target list y las cualificaciones sean finalizadas.
Esta función presenta un error de sintaxis: Las agregadas no pueden estar dentro de la cláusula where o las condiciones de un join.

**Archivo parse_func.c**

Maneja las llamadas de funciones en el parser (en este archivo se pueden encontrar numerosos mensajes de error pero la mayoría son de ejecución, no de sintaxis)

Incluye dentro de su código las siguientes librerías (entre otras):

postgres.h
parser/parse_coerce.h
parser/parse_expr.h
parser/parse_func.h
parser/parse_relation.h
parser/parse_type.h

Entre sus funciones se encuentran:

- **ParseFuncOrColumn**

En esta función se presentan diferentes tipos de evaluaciones hechas a las llamadas de funciones por ejemplo: revisa los números de parámetros permitidos por la función, revisa que no se ingresen valores nulos a las funciones, examina o chequea para hacer las proyecciones de columna, extrae la información de los arg type y transforma los argumentos RangeVar en varnodes de la manera apropiada.

- **match_argtypes**

Dada una lista de arreglos de typeid a una función y un arreglo de typeids de entrada, produce una lista corta de esos arreglos de typeid de funciones que son iguales a los typeid de entrada y retorna el número de dichos arreglos.

- **func_select_candidate**

Dado el arreglo de argtype de entrada y más de un candidato para la función, intenta resolver el conflicto. Retorna el candidato seleccionado si el conflicto puede ser resuelto, sino retorna NULL. Por diseño es similar a oper_select_candidate en el parse_oper.c

- **func_get_detail**

Intenta encontrar la función nombrada en los catálogos de sistema.
Si no se encuentra una igual:

1. chequea por posibles interpretaciones como un tipo de coerción trivial
2. consigue un vector de todos los posibles arreglos de arg type de entrada construidos desde las superclases del arg type de entrada original
3. consigue una lista de todos los posibles arreglos de tipo de argumentos a la función con nombre dado y número de argumentos
4. por cada arreglo de arg type de entrada desde el vector #1:
    a. encuentra cuantos arreglos de arg type de la función de la lista #2 pueden ser coaccionados
    b. si la respuesta es 1, se ha encontrado la función

c. si la respuesta es más de uno, intentar resolver el conflicto
d. si la respuesta es cero, intenta el siguiente vector desde el vector #1

También Retorna los tipos de argumentos verdaderos a la función.

- **argtype_inherit**

Construye un vector de arg type reflejando las propiedades heredadas de los argv suplidos. Esta función es usada para solucionar problemas de ambigüedad entre funciones con nombres iguales pero diferentes signaturas o firmas. Esta toma un arreglo de tipos de ids de entrada y retorna un vector de nuevos tipos de Oid.

- **make_arguments**

Dado el número y tipo de argumentos a una función, y los argumentos actuales y los tipos hace el CAST de tipos necesarios.

- **setup_field_select**

Crea un nodo FieldSelect que dice cual atributo será proyectado. Esta rutina es llamada por ParseFuncOrColumn() cuando necesita encontrar una proyección en un resultado de una función o un parámetro.

- **ParseComplexProjection**

Maneja los llamados de funciones con un solo argumento que es de tipo complejo.
Si el llamado de función es actualmente una proyección de columna, retorna un árbol de expresión convenientemente transformado, de lo contrario retorna NULL.

- **find_aggregate_func**

Rutina para chequear que una función exista y sea una agregada

- **LookupFuncName**

Dada un nombre de función de cualificación-posible y conjunto de tipos de argumentos buscar la función, retorna InvalidOid si la función no es encontrada

También está entre sus funciones todo el trabajo de herencia de argumentos, valida el mensaje de error "No such attribute" (Atributo no mencionado), presenta una función de tratamiento de errores.

**Archivo parser_nodes.c**

Compuesto por las rutinas encargadas de crear los nodos del querytree.

Entre sus funciones se encuentran:

- **ParseState * make_parsestate**

Crea e inicializa un ParseState. Muy utilizado en el analize.c, en todas las funciones de transformación, para pasar de un parsertree a un querytree y en casi todos los archivos parser_*.

- **Expr *make_op**

Transforma la expresión del operador asegurando compatibilidad de tipos. Utilizado por el parser_expr cuando el nodo es de tipo T_A_Expr y su campo expr es de tipo OPER.

- **Var *make_var**

Crea un nodo Var para un atributo identificado en el RTE y attrno.

- **ArrayRef *transformArraySubscripts**

Utilizado cuando intervienen expresiones o updates en la consulta. Algunas de las funciones que lo llaman son transformIndirection del parser_expr.c y updateTargetListEntry del parser_target.c

- **Const * make_const**

Convierte un valor del nodo (según lo retornado por la gramática) a un nodo Const del tipo "natural" para la constante. Esta rutina se usa solamente cuando no hay cast explícito para la constante, así que obtiene el tipo.

**Archivo parse_target.c**

Encargado de manejar los Target List.

Entre sus funciones se encuentran:

- **TargetEntry *transformTargetEntry**

Transforma cualquier nodo del tipo "expression-type" en una entrada del targetlist. Se exporta de modo que parse_clause.c pueda generar las entradas del targetlist para ORDER/GROUP BY, ítems que aun no están en el targetlist.
Utilizado en la función findTargetlistEntry de paser_clause.c

- **List *transformTargetList**

Cambia a una lista de ResTarget en una lista de TargetEntry. No tiene en cuenta si es SELECT, INSERT, o UPDATE solo transforma las expresiones dadas. Utilizado por el analize.c en las funciones transformInsertStmt, transformSelectStmt, transformUpdateStmt.

- **void updateTargetListEntry**

Se utiliza en declaraciones del INSERT y UPDATE solamente. Prepara un TargetEntry para asignarlo a una columna de target table. Cambia el valor dado al tipo de la columna del target (si es necesario).
Utilizado por el analize.c en las funciones transformInsertStmt, transformUpdateStmt.

- **List *checkInsertTargets**

Generan una lista de las columna del INSERT si no es provista, o prueba los nombres provistos, para cerciorarse que están en la columna de tabla indicada.

- **static List *ExpandAllTables**

Maneja los ' * ' (en target list) en una lista de las entradas del targetlist.

**ANALISIS OPTIMIZER**

Dentro del código fuente, el módulo de optimización de consultas, se encuentra en la ruta …usr/backend/optimizer/.

En este directorio se toma una estructura Query retornada por parser, y se genera un Plan que utiliza el executor. En este directorio se encuentran los siguientes subdirectorios:

- *optimizer/path***:** Toma la salida del Parser y genera todos los posibles métodos de ejecución de la solicitud. Examina el orden de unir las relaciones, las cláusulas Where de restricciones, y las estadísticas sobre las tablas, para evaluar cada posible método de ejecución, y asignar un costo a cada uno de estos.
- *optimizer/geqo***:** En optimizer/path, se generan todas las posibles combinaciones de Join entre las tablas. Cuando el número de tabla es grande, el número de Paths examinados se vuelve muy grande. El optimizador de consulta genético, hace una búsqueda semi-aleatoria a través del árbol de Joins, más que considerar exhaustivamente todos los posibles árboles de Join. Cada Join considerado por optimizer/geqo es pasado a optimizer/path para crear una ruta, así consideraremos todas las posibles PATH para cada par específico de Join, aun en modo GEQO. Para un número de tablas pequeño dentro de la consulta, este método consume mucho tiempo, pero para un número de tablas grande, el método es rápido.
- *optimizer/plan***:** Toma la salida de optimizer/path, escoge el Path con el menor costo, y crea un plan para el executor.
- *optimizer/prep***:** Realiza varios pasos de preprocesamiento para casos especiales.
- *optimizer/util***:** Contiene rutinas de soporte, utilizados por otras partes del optimizador.

El punto de entrada al proceso de optimización, es la función Planner, la cual toma una estructura Query y genera un Plan de consulta. Aquí se tienen en cuenta las siguientes funciones:

*Función Planner()*

- Prepara la estructura Query, para la manipulación recursiva de las subconsultas.
- Se apoya en la función subquery_planner.
- Realiza un proceso de limpieza en la estructura Plan, eliminando la información no requerida por el executor, después de realizar la optimización.

*Función subquery_planner()*

- Desplaza las subconsultas, contenidas en el campo rangetable, al nivel más externo del árbol. Si es posible simplifica expresiones constantes.
- Canoniza el campo qual. Intenta reducir cláusulas WHERE a las formas canónicas CNF (Forma Normal Conjuntiva) o DNF(Forma Normal Disyuntiva). La forma CNF es preferida, debido a que el optimizador puede utilizar cualquiera de las

subcláusulas AND para filtrar tuplas. Si el campo qual es de la forma o similar a DNF, sufrirá una expansión exponencial si se trata de convertirla a la forma CNF. En casos patológicos la transformación puede expandir el campo qual excesivamente; por lo tanto es preferible dejarla sin normalizar. Esto produce una reducción en la exactitud en los estimados de selectividad.

- Procesa Subenlaces.
- Convierte la estructura Vars de los niveles de consulta más externos a la estructura Params.
- Se apoya en la función grouping_planner.

*Función grouping_planner()*

- Preprocesa la lista del campo target, para consultas no SELECT.
- Manipula las cláusulas UNIÓN/INTERSECT/EXCEPT, GROUP BY, HAVING, funciones agregadas, ORDER BY, DISTINCT, LIMIT.
- Se apoya en la función query_planner.
- Se realiza agrupamiento(GROUP).
- Se efectúan las funciones agregadas.
- Se crea unique (DISTINCT).
- Se crea sort (ORDER BY).
- Se crea limit(LIMIT/OFFSET)

*Función query_planner()*

- Desplaza hacia fuera las constantes dentro del campo quals, que se pueden utilizar como entrada de ejecución de todo el Plan. Si se encuentra alguna constante, se crea un nodo Result en el nivel superior para hacer la entrada.
- Crea una lista target simplificada, que solo contiene nodos Vars, sin expresiones.
- Se apoya en la función subplanner.
- Se retorna las constantes del campo qual y la lista target no simplificada.

*Función subplanner()*

- Crea una lista de las relaciones base, utilizadas en la consulta.
- Separa las restricciones de la forma ( a =1 ) y los Join de la forma ( b=c ), dentro del campo qual.
- Encuentra cláusulas en el campo qual, que habilitan Join merge-sort y Hash-Join.
- Se apoya en la función make_one_rel, para obtener el plan primario.

*Función make_one_rel()*

- Se apoya en la función set_base_rel_pathlist
- Se apoya en la función make_fromexpr_rel.

*Función set_base_rel_pathlist()*

- Para cada relación base, se encuentra el Path de recorrido secuencial y los Path a través de todos los índices.
- Encuentra la selectividad de las columnas utilizadas en los Join.

*Función make_fromexpr_rel ()*

- Cambia el flujo del procesamiento de la consulta, al módulo GEQO, en caso de ser necesario. En caso contrario, se apoya en la función make_one_rel_by_joins () para invocar el método de optimización por defecto de PostgreSQL.

*Función make_one_rel_by_joins()*

- Encuentra todos los posibles Paths de la consulta, a través de una búsqueda sucesiva, de las formas de unir relaciones dentro una relación Join.
- Se apoya en la función make_rels_by_joins, para determinar todas las posibles parejas relaciones que serán unidas en este nivel, a partir de las parejas de unidas en el nivel anterior.
- Se apoya en la función set_cheapest() para extraer el Path más barato, para cada nuevo Join construido.
- Se verifica si no se está en el nivel superior del Join.

*Función make_rels_by_joins()*

- Para cada Join del nivel previo, si se tiene una cláusula Join, se ejecuta la función make_rels_by_clause_joins. En caso contrario, se ejecuta la función make_rels_by_clauseless_joins.
- Se genera un Plan denso, que une los Join de niveles inferiores.

## ANALISIS PATH

### Archivo allpaths.c

Rutinas para encontrar posibles paths para procesar un query así como un conjunto de rutinas que soportan la etapa de depuración del optimizer que se encargan de imprimir el contenido de las estructuras de datos solicitadas.

Incluye las siguientes librerías:

postgres.h
nodes/print.h
optimizer/clauses.h
optimizer/cost.h
optimizer/geqo.h
optimizer/pathnode.h
optimizer/paths.h
optimizer/plancat.h
optimizer/planner.h
optimizer/prep.h
parser/parsetree.h
rewrite/rewriteManip.h

Entre sus funciones se encuentran:

- **make_one_rel**

Encuentra todos los paths de acceso posibles para ejecutar un query, retornando un rel que representa el join de todos los rels base en el query.

- **set_base_rel_pathlists**

Encuentra todos los paths disponibles para el escaneo de cada relación base de entrada. Escaneo secuencial y cualquier índice disponible son considerados. Cada path útil se une al campo 'pathlist' de su relación.

- **set_plain_rel_pathlist**

Construye los paths de acceso para un plan (ningún subquery, ninguna herencia).

- **set_inherited_rel_pathlist**

Construye paths de acceso para una rel heredada. Inheritlist es una lista de indexs RT de todas las tablas en el árbol de la herencia, incluyendo un duplicado del mismo padre.

- **set_subquery_pathlist**

Construye solamente el path de acceso para un subquery RTE.

- **set_function_pathlist**

Construye solamente el path de acceso para una función RTE.

- **make_fromexpr_rel**

Construye los paths de acceso para un nodo jointree de FromExpr.

- **make_one_rel_by_joins**

Encuentra todos los joinpaths posibles para un query, encontrando maneras para unir relaciones en joins. Retorna la relación que es el resultado de unir todas las relaciones originales juntas.

- **subquery_is_pushdown_safe**

Subquery es el componente particular que será comprobada. Topquery es el componente superior de un árbol de set-operations (el mismo query si no está implicado set-op). Las siguientes condiciones son comprobadas aquí:

1. Si el subquery tiene una cláusula de LIMIT o un DISTINCT ON, no se debe empujar hacia abajo cualquier quals, ya que podría cambiar el conjunto de filas retornado.

2. Si el subquery tiene cualquier función de retorno en su target list, no empuja hacia abajo ningún quals, puesto que los quals puede referir a esos ítems del target list, lo que siguiente significaría que se introduciría funciones de retorno en los quals WHERE/HAVING de los subquery.

3. Si el subquery contiene EXCEPT o EXCEPT ALL no puede empujar los quals, porque eso cambiará los resultados. Para los subqueries con UNION/UNION ALL/INTERSECT/INTERSECT ALL, se puede empujar los quals en cada componente query, siempre y cuando todo el componente query comparta tipos idénticos de la salida.

- **recurse_pushdown_safe**

Rutina de ayuda para trabajar con los setOperations del árbol.

- **subquery_push_qual**

Empuja hacia abajo un qual que se ha determinado como seguro.

- **recurse_push_qual**

Rutina de ayuda para los setOperations del árbol.

**Archivo clausesel.c**

Rutinas para computar o calcular las selectividades de cláusula.

Incluye las siguientes librerías:

postgres.h
catalog/pg_operator.h
catalog/pg_type.h
nodes/makefuncs.h
optimizer/clauses.h
optimizer/cost.h
optimizer/plancat.h
optimizer/restrictinfo.h
parser/parsetree.h
utils/fmgroids.h
utils/lsyscache.h
utils/selfuncs.h

Define las siguientes estructuras:
RangeQueryClause: Estructura de datos para acumular información acerca de posibles pares de cláusulas range-query en clauselist_selectivity

Entre sus funciones se encuentran:

- **Rutinas para calcular selectivities**

- **restrictlist_selectivity**

Calcula la selectividad de una lista implicitly-ANDed de RestrictInfo clauses. Esta es la misma clauselist_selectivity excepto por la representación de la lista de cláusulas

- **clauselist_selectivity**

Calcula la selectividad de una lista implicitly-ANDed de cláusulas de expresión booleana. La lista puede estar vacía, donde 1.0 debe ser retornado. Observe clause_selectivity() para el siguiente significado o sentido del parámetro varRelid.
La Aproximación básica es tomar el producto de las selectividades de las subcláusulas. No obstante, eso solo esta bien si las subcláusulas tienen probabilidades independientes, y en realidad son frecuentemente no independientes.
Seguidamente, lo único listo que se tiene es reconocer "range queries", tal como "x > 34 AND x < 42". Las cláusulas son reconocidas como posibles componentes range query si ellas son opclauses de restricción cuyas operaciones tienen scalarltsel() o scalargtsel() como sus estimadores de selectivas de restricción. Se aparea cláusulas de esta forma que refiere a la misma variable. Una cláusula no-apareable de este tipo es simplemente multiplicada dentro del producto de la selectividad de manera normal. Pero cuando se encuentra un par, se sabe que la selectiva representa las posiciones relativas de los bajos y altos bounds con

los rangos de columnas, en lugar de calcular la selectividad como hisel * losel, se puede calcularlo como hisel + losel -1. (Para mirar esto, note que hisel es la fracción del rango debajo del bound alto, mientras losel es la fracción por encima del bound bajo; hisel puede ser interpretado directamente como un valor 0..1 pero necesitamos convertir losel a 1-losel antes de interpretarlo como un valor. Luego el rango disponible es 1-losel a hisel. No obstante, este cálculo doble-excluye nulos, entonces realmente se necesita hisel + losel + null_frac - 1.)
Si el cálculo produce cero o negativo, como sea, se usa el estimado por defecto; eso probablemente signifique que una o ambas selectivas son un estimativo por defecto antes que un range value actual. Por supuesto esto es totalmente dependiente del comportamiento de scalarltsel/scalargtsel; quizás algún día se pueda generalizar la aproximación.

- **addRangeClause**

Adhiere un nuevo range clause para clauselist_selectivity. Aqui es donde se trata de igualar pares de cláusulas range-query.

- **clause_selectivity**

Calcula la selectividad de una cláusula de expresión booleana general. varRelid es 0 o un índice de rangetable. Cuando varRelid no es 0, solo variables pertenecientes a esa relación son consideradas en el cálculo de la selectiva; otras vars son tratadas como constantes de valores desconocidos. Esto es apropiado para estimar la selectividad de una cláusula join que esta siendo usada como una cláusula de restricción en un escaneo de un nestloop join's inner relation --- varRelid debería ser el ID de la relación inner. Cuando varRelid es 0, todas las variables son tratadas como variables. Esto es apropiado para cláusulas join ordinarias y cláusulas de restricción.

**Archivo costsize.c**

Rutinas para computar (y el sistema) tamaños de la relación y los costes de la trayectoria.

Los costes de la ruta se miden en unidades de accesos a disco: una página secuencial buscada tiene costo 1. Todo se escala dependiendo de la página buscada, con los parámetros de escalamiento

| | |
|---|---|
| random_page_cost | costo de una página no secuencial buscada |
| cpu_tuple_cost | costo de tiempo típico de la CPU para procesar una tupla |
| cpu_index_tuple_cost | costo del tiempo típico de la CPU para procesar una tupla índice. |
| cpu_operator_cost | costo de tiempo de la CPU para procesar un típico operador WHERE. |

También se utiliza un cálculo aproximado "effective_cache_size" del número de páginas disco en Postgres + OS-nivel de cache de disco (no se puede utilizar simplemente NBuffers para este propósito porque ése no haría caso de los efectos de cache de disco del kernel).

Obviamente, tomar constantes para estos valores es una sobre simplificación, pero es bastante difícil conseguir cualquier estimación útil uniforme a este nivel de detalle. Observe que todos estos parámetros son user-settable, en caso de que los valores predefinidos están drásticamente apagados para una plataforma particular.

Se computa dos costes separados para cada ruta:

total_cost: costo total estimado para traer todas las tuplas.

startup_cost: costo consumido antes que la primera tupla es traída.

En algunos escenarios, tales como cuando hay un LÍMIT o se esta implementando un EXISTS(...) (sub-select), no es necesario traer todas las tuplas del resultado de la ruta. Un llamador puede estimar el coste de traer un resultado parcial interpolando entre el startup_cost y el total_cost. Detalladamente:

actual_cost = startup_cost + (total_cost - startup_cost) * tuples_to_fetch/path->parent->rows;

Observe que la cuenta de las filas de una relación base (y, por la extensión, los plan_rows para los nodos del plan debajo del nodo LÍMIT) están fijados sin consideración alguna hacia cualquier LÍMIT, así que esta ecuación trabaje correctamente (también, estas rutinas garantizan no colocar el contador de filas a cero, así no será dividido por cero). El LÍMIT es aplicado como nodo a nivel superior del plan.

Por razones en gran parte históricas, la mayoría de las rutinas en este módulo usan la path resultante pasada para almacenar solamente sus resultados startup_cost y total_cost.
Todos los datos de entrada que necesitan se pasan como parámetros separados aunque mucho de este se podría extraer de la path.

Una excepción se hace para las rutinas del cost_XXXjoin(), que cuentan con todos los campos sin-costo del XXXPath pasado que se completará.

Incluye las siguientes librerías:

postgres.h
math.h
catalog/pg_statistic.h
executor/nodeHash.h
miscadmin.h
optimizer/clauses.h
optimizer/cost.h
optimizer/pathnode.h
optimizer/plancat.h
parser/parsetree.h
utils/selfuncs.h
utils/lsyscache.h
utils/syscache.h

Entre sus funciones se encuentran:

- **cost_seqscan**

Determina y retorna el costo de explorar una relación secuencialmente.

- **cost_nonsequential_access**

Estima el coste de tener acceso a una página al azar de una relación (o clasifique el archivo temporal) del tamaño dado en páginas.

El modelo simplista que el costo es random_page_cost es lo que se desea utilizar para las relaciones grandes; pero para los pequeños es un serio sobrestimación debido a los efectos de caching. Esta rutina intenta explicar eso.

Desafortunadamente no se tiene una buena manera de estimar el tamaño eficaz de caché.
Se sabe que Postgres tiene NBuffers almacenadores intermedios internos, pero el tamaño del caché de disco del kernel es incierto, y cuánto de él se consigue utilizar es incluso menos seguro. Por ahora se asume que se da un parámetro effective_cache_size.

Dado un supuesto tamaño de cache, se estima el costo I/O actual por página con las ecuaciones enteramente ad-hoc:

```
if relpages > = effective_cache_size:
        random_page_cost * (1 - (effective_cache_size/relpages)/2)
if relpages < effective_cache_size:
        1 + (random_page_cost/2-1) * (relpages/effective_cache_size) ** 2
```

Estos dan el comportamiento asintótico derecho (=> 1,0 mientras que los relpages sean pequeños, => random_page_cost mientras que llega a ser grande) y encuentran en el centro con la estimación que el cache es acerca de 50% eficaz para una relación del mismo tamaño que effective_cache_size.

- **cost_index**

Determina y retorna el coste de explorar una relación usando un índice.

NOTA: un nodo indexscan del plan puede representar realmente varios pasos, pero aquí se considera el coste de apenas un paso.

'raíz' es la raíz del query
'baserel' es la relación base de índice
'índice' es el índice que se utilizará
'indexQuals' es la lista de las cláusulas qual aplicables (implícitas y semántica)
'is_injoin' es T si se esta considerando usar la exploración del índice como el interior de un nestloop join (por lo tanto, algunos de los indexQuals son cláusulas join)

NOTA: 'indexQuals' debería contener solamente las cláusulas usables como restricciones del índice. Cualquier quals adicional evaluado como qpquals puede reducir el número de tuplas retornadas, pero no reducirán el número de tuplas que tenemos que traer de la tabla, así no reduce el coste de la exploración.

NOTA: para estimar el número de tuplas y páginas traídas de la tabla principal utiliza: una aproximación propuesta por Mackert y Lohman, el "Index Scans Using a Finite LRU Buffer: A Validated I/O Model", transacciones de ACM en sistemas de la base de datos, el vol. 14, No. 3, septiembre de 1989, pagina 401-424.

- **cost_tidscan**

Determina y retorna el coste de explorar una relación usando TIDs.

- **cost_subqueryscan**

Determina y retorna el costo de examinar una subquery RTE.

- **cost_functionscan**

Determina y retorna el coste de explorar una función RTE.

- **cost_sort**

Determina y retorna el coste de clasificar una relación, incluyendo el coste de leer los datos de entrada.

Si el volumen total de datos a clasificar es menos que SortMem, se hará una clasificación en-memoria, que no requiere ningún I/O y alrededor de t*log2(t) comparaciones de tuple para t tuples.

Si el volumen total excede SortMem, se cambia a un algoritmo de unión tape-style. Todavía habrá cerca de t*log2(t) comparaciones del tuple en total, pero también se necesitará escribir y leer cada tuple una vez por paso de la unión. Se espera cerca de ceil(log6(r)) la pasos de unión donde r es el número de recorridos iniciales formados (log6 porque tuplesort.c usa la combinación six-tape).
Puesto que el promedio de recorrido inicial debe estar sobre dos veces SortMem, tenemos:

disk traffic = 2 * relsize * ceil(log6(p/(2*SortMem)))
CPU = comparison_cost * t * log2(t)

El disk traffic se asume que es acceso medio secuencial y medio al azar.

Se carga dos evals del operador por comparación de tuple, que debe estar en el ballpark derecho en la mayoría de los casos.

'pathkeys' son una lista de las claves de ordenación
'input_cost' es el coste total para leer los datos de entrada
'tuples' son el número de tuplas en la relación
'anchura' es el ancho promedio en bytes

NOTA: Algunos llamadores pasan NIL para los pathkeys porque ellos no puede proveer convenientemente las claves de ordenación. Puesto que esta rutina actualmente hace cualquier cosa con los pathkeys, de todos modos, ésa no importa pero si lo hace siempre, él debe reaccionar a la carencia de los datos claves.

- **cost_material**

Determina y retorna el coste de materializar una relación, incluyendo el coste de leer los datos de entrada.

Si el volumen de datos total a materializar excede SortMem, se necesitará escribir al disco, así que el coste es mucho más alto en ese caso.

- **cost_agg**

Determina y retorna el coste de realizar un nodo Agg del plan, incluyendo el coste de su entrada.

Nota: cuando es aggstrategy == AGG_SORTED, llamador debe asegurarse de que los costes de la entrada son de la entrada appropriately-sorted.

- **cost_group**

Determina y retorna el coste de realizar un nodo Group plan, incluyendo el coste de su entrada.

Nota: el llamador debe asegurarse de que los costes de entrada son de la entrada appropriately-sorted.

- **cost_nestloop**

Determina y retorna el coste de combinar dos relaciones usando el algoritmo nested loop. 'path' es llenado ya en excepción por los campos de coste.

- **cost_mergejoin**

Determina y retorna el costo de combinar dos relaciones usando el algoritmo merge join. 'path' es llenado ya en excepción por los campos de coste.

Nota: los mergeclauses de los paths debería ser un subconjunto de la lista joinrestrictinfo; outersortkeys y innersortkeys son listas de los keys a ser usados para clasificar las relaciones externas e internas, o NIL si no es explícito; sort es necesario porque la path fuente es ya ordenada.

- **cost_hashjoin**

Determina y retorna el costo de combinar dos relaciones usando el algoritmo hash join. 'path' es llenado ya en excepción por los campos de coste.

Note: los hashclauses de los path deben ser un subconjunto de la lista joinrestrictinfo.

- **Estimate hash bucketsize** fraction (ie, número de entradas en un bucket dividido por el total de tuplas en la relación) si el Var especificado es usado como un hash key.

- **cost_qual_eval**

Estima el costo de CPU de evaluar una cláusula WHERE. La entrada puede ser cualquier de la lista implicitly-ANDed de expresiones booleanas, o una lista de nodos RestrictInfo. El resultado incluye ambos un componente one-time (startup), y un componente per-evaluation.

- **approx_selectivity**

Estimación rápida y sucia de cláusulas selectivities. La entrada puede ser cualquier lista implicitly-ANDed de expresiones booleanas, o una lista de nodos RestrictInfo.

La parte "rápida" viene de coger las estimaciones de selectividad, así podemos evitar recomputarlos después.

La parte "sucia" viene del hecho que las selectivities de cláusulas múltiples son estimadas independientemente y multiplicadas a la vez. Ahora con frecuencia clauselist_selectivity no puede hacer algo mejor de todos modos, pero es astuto para algunas situaciones.

- **set_baserel_size_estimates**

Fija las estimaciones de tamaño para la relación base dada.

Las listas targetlist y restrictinfo de las relaciones deberían ya haber sido construidas.

Se agrupa los siguientes campos del nodo rel:

rows: el número estimado de tuplas de salida (después aplicando cláusulas de restricción).
width: el promedio estimado de tuplas de salida en bytes.
baserestrictcost: costo estimado de evaluar cláusulas baserestrictinfo.

- **set_joinrel_size_estimates**

Fija las estimaciones de tamaño de la relación join dado.

Los targetlist de las relaciones ya deberían haber sido construidas y una lista de cláusulas de restricción que empareje los componentes de relaciones dados.

Puesto que hay más de una forma para hacer un joinrel para más de dos relaciones base, los resultados que se consigue aquí podría depender sobre cual componente del par de rel es provista. En teoría no se debe conseguir las mismas respuestas cual par es proporcionada; en la práctica, desde las rutinas de estimación de selectividad no manejan todos los casos igualmente bien.

Es importante que los resultados para JoinTypes simétrico sea simétrico, eg, (rel1, rel2, JOIN_LEFT) debe producir el mismo resultado que (rel2, rel1, JOIN_RIGHT). También, JOIN_IN debe producir el mismo resultado que JOIN_UNIQUE_INNER, así mismo de JOIN_REVERSE_IN == JOIN_UNIQUE_OUTER.

Se fija los mismos campos del relno de que lo hace el set_baserel_size_estimates().

- **set_function_size_estimates**

Fija las estimaciones de tamaño para una relación base que es una llamada a función.

Las listas targetlist y restrictinfo de las relaciones deberían ya haber sido construidas.
Se agrupó los siguientes campos del nodo rel:
rows: el número estimado de tuplas de salida (después aplicando cláusulas de restricción).
width: el promedio estimado de tuplas de salida en bytes.
baserestrictcost: costo estimado de evaluar cláusulas baserestrictinfo.

- **set_rel_width**

Fija el ancho de salida estimado de una relación base.

NOTA: Esto trabaja lo mejor posible en relaciones sencillas porque prefiere mirar Vars reales. No podrá hacer uso del info pg_statistic cuando se aplique a una relación del subquery.

La estimación de ancho de per-attribute son tomados para posibles reutilizaciones mientras se estructura las relaciones join.

- **relation_byte_size**

Estima el espacio de almacenamiento en bytes para un número de tuplas dadas de un ancho dado (tamaño en bytes).

- **page_size**

Retorna una estimación del número de páginas cubiertas por un número de tuplas dadas de un ancho dado (tamaño en bytes).

**Archivo indxpath.c**

Rutinas para determinar cuales índices son de uso para escanear una relación dada, y crear IndexPaths de acuerdo con ellos.

Incluye las siguientes librerías:

access/heapam.h
access/nbtree.h
catalog/catname.h
catalog/pg_amop.h
catalog/pg_namespace.h
catalog/pg_operator.h
executor/executor.h
nodes/makefuncs.h
nodes/nodeFuncs.h
optimizer/clauses.h
optimizer/cost.h
optimizer/pathnode.h
optimizer/paths.h
optimizer/restrictinfo.h
optimizer/var.h
parser/parse_coerce.h
parser/parse_expr.h
parser/parse_oper.h
rewrite/rewriteManip.h
utils/builtins.h
utils/fmgroids.h
utils/lsyscache.h
utils/selfuncs.h
utils/syscache.h

Entre sus funciones se encuentran:

- **create_index_paths**

Genera todos los index paths interesantes para la relación dada. Paths candidatos son añadidos a la pathlist de la rel (usando add_path). Nodos IndexPath adicionales pueden también ser añadidos a la lista innerjoin de la rel.

Para ser considerado para un index scan, un index debe coincidir a una o más restrictin clauses o join clauses de las qual condition de la consulta (query), o coincidir la condición ORDER BY de la query.
Hay dos clases básicas de index scans. Un index scan "plano" usa solamente restriction clauses (posiblemente ninguna) en su indexqual, así que puede ser aplicado en cualquier

contexto. Un "innerjoin" index scan usa join clauses (además de restriction clauses, si están disponibles) en su indexqual.
Por consiguiente puede ser usado solamente como el inner relation de un nestloop join contra un outer rel que incluye todas las otras rels mencionadas en sus join clauses. En ese contexto, los valores para los atributos de las otras rels están disponibles y reparados durante cualquier scan del indexpath.

Un IndexPath es generado y enviado al add_path() por cada index (índice); esta rutina se considera potencialmente interesante para la query actual.
Un innerjoin path es también generado para cada combinación interesante de relaciones outer join. Los paths innerjoin *no* son pasados al add_path(), pero son agregados (append) a la lista "innerjoin" de la relación para consideraciones posteriores en nested-loop joins.

'rel' es la relación para la cual se quiere generar los index paths.

- **RUTINAS PARA PROCESAR 'OR' CLAUSES**

- **match_index_orclauses**

Procura coincidir un index contra subclauses con 'or' clauses. Cada subclause que coincide es marcada con el nodo del index.

Esencialmente, esto añade el 'index' a la lista de subclause índices en el campo RestrictInfo de cada una de las cláusulas 'or' donde este coincida.
Se puede usar el almacenamiento en el RestrictInfo para este propósito porque este procesamiento es hecho solamente sobre restriction clauses de una relación-sencilla.
Por lo tanto, nunca se tendrá índices para más de una relación mencionada en la misma lista de RestrictInfo node.

'rel' es el nodo de la relación sobre la cual el index es definido.
'index' es el nodo index .
'restrictinfo_list' es la lista de restriction clauses disponibles.

- **match_index_orclause**

Procura coincidir un index contra subclauses de una 'or' clause.

Una coincidencia significa que:
1. El operador con la subclause puede ser usado con el operador clase especificado del index, y
2. Un operando de la subclause coincide con el index key.

Si una subclause es una 'and' clause, entonces esta coincide si cualquiera de sus subclauses es una opclause que coincida.

'or_clauses' es la lista de subclauses con la 'or' clause.
'other_matching_indices' es la lista de información sobre otros índices que ya han coincidido con subclauses con esta.
'or' clause en particular (i.e. una lista previamente generada por esta rutina), o NIL si esta rutina no ha corrido previamente para esta 'or' clause.

Retorna una lista de la forma ((a b c) (d e f) nil (g h) ...) donde a,b,c son nodos de índices que coinciden con la primer subclause en 'or-clauses', d,e,f coinciden con la segunda subclause, ningún índice coincide con la tercera, g,h coinciden a la cuarta, etc.

- **match_or_subclause_to_indexkey**

Mira si una subclause de un OR clause coincide con un index.

Se acepta la subclause si esta es un operator clause que coincide con el index, o si este esta es una AND clause y cualquiera de cuyos miembros es una opclause que coincide con el index.

Para multi-key indexes, solamente se busca coincidencias para la primera key; sin la cual una coincidencia de index es inutil. Si la clause es una AND entonces se puede extraer subclauses adicionales para usar con los posteriores indexkeys, pero no hay que preocuparse por eso hasta que extract_or_indexqual_conditions() sea llamado.

- **extract_or_indexqual_conditions**

Dada una subclause OR que ha sido previamente determinada para coincidir a un index específico, extrae una lista de opclauses específicas que pueden ser usadas como indexquals.

En el caso más simple significa hacer una lista de un elemento de la opclause dada. Sin embargo, si la subclause OR es una AND, se tiene que escanear para encontrar opclause(s) que coincidan con el index (Debe haber al menos una, if match_or_subclause_to_indexkey exitosa, pero puede haber mas).

También, pueden verse otras restriction clauses de la relación para descubrir indexquals candidatos adicionales: por ejemplo, considera ... where (a = 11 or a = 12) and b = 42; si se esta tratando con un index sobre (a,b) entonces se puede incluir la clause b = 42 en la lista indexqual generada para cada una de las subclauses OR.
Esencialmente se hace una transformacion index-específica desde CNF hacia DNF.
También se aplica expand_indexqual_conditions() para convertir cualquier coincidencia especial de opclauses a operadores indexables.

La cláusula passed-in no cambia.

- **ROUTINAS PARA CHEQUEAR RESTRICCIONES**

- **group_clauses_by_indexkey**

Invocada por la función principal. Generan una lista de restriction clauses que pueden ser usadas con un index.

- **group_clauses_by_ikey_for_joins**

Invocada por funciones que trata con joins. Genera una lista de join clauses que pueden ser usadas con un index para escanear el inner side de un nestloop join.

- **match_clause_to_indexkey**

Llamada por todas las funciones superiores. Determina si una restriction o join clause coincide con una key de un index.

- **indexable_operator**

Verifica si la clause binaria contiene un operador de la clase Oid que se necesita

- **RUTINAS QUE HACEN TEST PARCIAL AL INDEX PREDICATE**

Las siguientes funciones horadan en el predicado, revisan las clauses en restrictinfo_list en profundidad para ver la dependencia del resultado Verdad o Falsedad de las clauses. Si es un subgrupo AND, todas deben ser V, y si es OR con una V será suficiente. Es una búsqueda iterativa en profundidad.

- **pred_test**
- **pred_test_restrict_list**
- **pred_test_recurse_clause**
- **pred_test_recurse_pred**
- **pred_test_simple_clause**

- **RUTINA PARA CHEQUEAR JOIN CLAUSES**

- **indexable_joinclauses**

Encuentra todos los grupos de join clauses desde 'joininfo_list' que pueden ser usados en conjunción con 'index' para el inner scan de un nestjoin.

- **PATH CREATION UTILITIES**

- **index_innerjoin**

Create nodos index path correspondientes a los paths que serán usados como inner relations en nestloop joins.

- **CHECK OPERANDS**

- **match_index_to_operand**

Test generalizado para coincidencias entre una key del index y el operando sobre un lado de una restriction o join clause.

- **function_index_operand**

Función de apoyo para la anterior.

- **RUTINAS PARA OPERADORES INDEXABLES "ESPECIALES"**

- **match_special_index_operator**

Debido a operadores especiales que el executor no podrá comprender fácilmente

- **expand_indexqual_conditions**

Dada una lista de (ANDed implicitos) indexqual clauses, expande cualquier index operator "especial" en clauses que la maquinaria indexscan sabrá manejar. Las Clauses que no son reconocidas por match_special_index_operator() pasan sin cambio.

- **prefix_quals**

- **network_prefix_quals**

Estas dos funciones revisan prefijos de los operadores para verificar el tipo o clase de operación a realizar.

- **FUNCIONES DE APOYO**

- **find_operator**
- **string_to_datum**
- **string_to_const**

**Archivo joinpath.c**

Rutinas para encontrar todos los path's posibles para el procesamiento de un conjunto de joins.

Incluye las siguientes librerías:

postgres.h
math.h
optimizer/clauses.h
optimizer/cost.h
optimizer/pathnode.h
optimizer/paths.h
parser/parsetree.h
utils/lsyscache.h

Entre sus funciones se encuentran:

- **add_paths_to_joinrel**

Dada una relación join y dos componentes rels desde el cual esto puede ser hecho, considera todos los paths posibles que usan los dos componentes rels como rel outer e inner respectivamente. Adiciona esos paths al pathlist de los join rel si ellos pasan la comparación con otros paths (y remueven cualquier path existente que este dominado por esos paths). Modifica el campo pathlist del nodo joinrel para contener los mejores paths encontrados muy lejos.

- **sort_inner_and_outer**

Crea los paths de un join mergejoin organizando explícitamente ambas relaciones la outer e inner join en cada ordenamiento merge disponible.

- **match_unsorted_outer**

Crea los posibles join paths para procesar una relación join simple 'joinrel' empleando cualquier sustitución iterativa o mergerjoin en cada uno de estos outer paths (considerando solo outer paths que estén ya ordenados suficientemente bien para merging).

- **match_unsorted_inner**

Genera paths mergejoin que usan una organización explícita del outer path con un inner path ya organizado

- **hash_inner_and_outer**

Crea join paths para hashjoin hashing explícitamente ambas relaciones join la outer e inner de cada cláusula hash disponible

- **best_innerjoin**

Encuentra el index path más barato que ya ha sido identificado por indexable_joinclauses() como un posible inner path para la relación(es) outer dada en el join nestloop.

- **select_mergejoin_clauses**

Selecciona las cláusulas mergerjoin que son usables para un join particular. Retorna una lista de nodos RestrictInfo por esas cláusulas.

**Archivo joinrels.c**

Rutinas para determinar cuales relaciones deberían ser combinadas (joined).

Incluye las siguientes librerías:

postgres.h
optimizer/pathnode.h
optimizer/paths.h

Entre sus funciones se encuentran:

- **make_rels_by_joins**

Considerar caminos para producir relaciones join conteniendo exactamente el 'level' jointree ítems (Este es un paso del método de programación dinámica incorporado en make_one_rel_by_joins). Los nodos join rel factibles por cada combinación de relaciones de mas bajo nivel son creados y retornados en una lista.
Las rutas de implementación son creadas por cada joinrel semejante, también.

- **make_rels_by_clause_joins**

Estructura joins entre la relación dada 'old_rel' y otras relaciones que son mencionadas dentro de los nodos joininfo de old_rel's (i.e., relaciones que participan en cláusulas join que 'old_rel' también participa).
Los nodos join rel son retornados en una lista.

'old_rel' es la relación entrante por la relación a ser combinada.
'other_rels' otras rels a ser consideradas por joining.

- **make_rels_by_clauseless_joins**

Dando una relación 'old_rel' y una lista de otras relaciones 'other_rels', crea un relación join entre 'old_rel' y cada miembro de 'other_rels' que no esta ya incluido en 'old_rel'.
Los nodos join rel se retornan en una lista.

'old_rel' es la relación entrante por la relación a ser combinada.
'other_rels' otras rels a ser consideradas por joining.

Comúnmente, esto es únicamente usado con relaciones iniciales en other_rels, pero debería trabajar para joining a joinrels también.

- **is_inside_IN**

Detecta si la relación especificada tiene interno un IN (sub-SELECT).

- **make_jointree_rel**

Halla o estructura una combinación de relación RelOptInfo representando un ítem jointree específico. Para JoinExprs, únicamente se considera la construcción de la ruta que corresponda exactamente a lo que el usuario escriba.

- **make_join_rel**

Halla o crea un join RelOptInfo que representa el join de dos rels dadas, y agrega al path información de paths creados con las dos rels como relación outer e inner.
(El join rel podría ya contener paths generados de otros pares de rels que agregan igual conjunto de base rels.)

NB: retornar NULL si el join prueba no es válido. Esto puede únicamente pasar cuando trabaje con cláusulas IN que tengan joins internos.

**Archivo orindxpath.c**

Rutinas para encontrar paths de índices que asemejan un conjunto de cláusulas 'or'.

Incluye las siguientes librerías:

postgres.h
optimizer/cost.h
optimizer/pathnode.h
optimizer/paths.h
optimizer/restrictinfo.h

Entre sus funciones se encuentran:

- **create_or_index_paths**

Crea paths para índices que asemejan cláusulas 'or'. create_index_paths() ya debe haber sido llamado. 'rel' es la relación de entrada para la cual los paths están siendo creados. No retorna nada, pero adhiere paths al rel->pathlist via add_path().

- **best_or_subclause_indices**

Determina el mejor índice a ser usado en conjunto con cada subcláusula de una cláusula 'or' y el costo de escanear una relación usando esos índices. El costo es la suma de los costos de índices individuales, desde que el ejecutor realice un escaneo por cada subcláusula del 'or'. Retorna una lista de nodos IndexOptInfo, uno por escaneo.
Esta rutina también crea la lista indexqual que será necesitada por el ejecutor. La lista indexqual tiene una entrada por cada escaneo de la rel base, la cual es una sublista de condiciones indexqual a aplicar en el escaneo. Las semánticas implícitas son AND a través de cada sublist de quals, y OR a través de la lista de nivel tope (note que el ejecutor toma cuidado de no retornar cualquier tupla simple más de una vez).

'rel' es el nodo de la relación en la cual los índices son definidos.
'subclauses' son las subcláusulas de la cláusula 'or'.
'índices' es una lista de sublistas de los nodos IndexOptInfo que semejaron cada subcláusula de la cláusula 'or'.
'pathnode' es el nodo IndexPath a ser construido.

Los resultados son retornados por colocación de estos campos del pathnode repasado:
'indexinfo' siguientes en una lista del índice de nodos IndexOptInfo, uno por escaneo.
'indexqual' siguientes en los indexqual construidos por el path (una lista de sublistas de cláusulas, una lista por escaneo de la rel base).
'startup_cost' y 'total_cost' consiguientes en los costos de path completos.
'startup_cost' es el costo inicial solo para el primer index scan; startup costs para escaneos posteriores serán pagados después, ellos solo consiguen influir en el total_cost.

Nota: Se escoge cada escaneo en base de su costo total, ignorando el costo inicial. Esto es razonable siempre que todos los tipos de índice tengan cero o costos iniciales pequeños.

- **best_or_subclause_index**

Determina cual es el mejor índice a ser usado con una subcláusula de una cláusula 'or' por estimación de costo de uso de cada índice y seleccionando el menos costoso (considerando solo el costo total, por ahora).

'rel' es el nodo de la relación en la cual el índice es definido.
'subclauses' es la subcláusula OR a ser considerada.
'índices' es una lista de nodos IndexOptInfo que semejan la subcláusula.

Estas son las variables de retorno:

'retIndexInfo' consiguientesue el IndexOptInfo del mejor índice.
'retIndexQual' consiguientesue una lista de condiciones indexqual para el mejor índice.
'retStartupCost' consiguientesue el costo inicial de un escaneo con ese índice.
'retTotalCost' consiguientesue el costo total de un escaneo con ese índice.

**Archivo pathkeys.c**

Conjunto de utilidades para emparejar y construir path keys. En este archivo se pueden detallar las siguientes secciones PATHKEY COMPARISONS, NEW PATHKEY FORMATION, PATHKEYS AND SORT CLAUSES, PATHKEYS AND MERGECLAUSES y PATHKEY USEFULNESS CHECKS, donde la última sección se encarga de asegurar que el add_path() no considere un path para tener diferentes sort a menos que realmente sea útil, es decir estas rutinas se comprueban para saber si hay utilidad de los pathkeys dados.

Incluye las siguientes librerías:

postgres.h
nodes/makefuncs.h
optimizer/clauses.h
optimizer/pathnode.h
optimizer/paths.h
optimizer/planmain.h
optimizer/tlist.h
parser/parsetree.h
parser/parse_func.h
utils/lsyscache.h

Entre sus funciones se encuentran:

- **makePathKeyItem**

Crea un nodo PathKeyItem.

- **add_equijoined_keys**

La cláusula dada tiene un operador mergejoin, así que sus dos lados se pueden considerar iguales después de la aplicación de la cláusula de la restricción; en detalle, cualquier pathkey que menciona un lado (con el sortop correcto) puede ser ampliado para incluir el otro lado también. Registra los vars y los sortops asociados en el equi_key_list del query para el uso futuro.

- **generate_implied_equalities**

Escanea el equi_key_list terminado para el query y genera calificaciones (WHERE clauses) para todas las parejas igualdades aun no mencionadas en los quals. Esto es útil porque las cláusulas ayudan al código selectivity-estimación, y es necesario para asegurarse de que las sort keys sean realmente equivalentes.
Esta rutina apenas recorre el equi_key_list para encontrar todas las parejas igualdades. Llama a process_implied_equality (plan/initsplan.c) para determinar si cada ítem ya se conoce, y agregarlo a la lista apropiada del restrictinfo en caso de que no sea así.

- **make_canonical_pathkey**

Dado un PathKeyItem, encuentra el subconjunto del equi_key_list que sea miembro de él. Si es así retorna un puntero a esa sublista, que es una representación canónica de ese set equivalencia del PathKeyItem. Si no se encuentra, agregua un singleton "equivalence set" al equi_key_list y lo retorna (Mire compare_pathkeys).
Esta función no se debe utilizar hasta después de que se haya terminado el escaneo de la cláusula WHERE para los operadores del equijoin.

- **canonicalize_pathkeys**

Convierte los pathkeys a la forma canónica. Esta función no se debe utilizar hasta después de que se haya terminado el escaneo de la cláusula WHERE para los operadores del equijoin.

- **compare_pathkeys**

Compare dos pathkeys para ver si son equivalentes, y si no, si uno es "mejor" que el otro. Esta función se puede aplicar solamente a las listas canonizadas del pathkey. En la representación canónica, las sublistas se pueden comprobar para saber si hay igualdad por la comparación simple del puntero.

- **compare_noncanonical_pathkeys**

Compara dos pathkeys para ver si son equivalentes, y si no, si uno es "mejor" que el otro. Se usa cuando se debe comparar pathkeys no-canonizados. Un pathkey se puede considerar mejor que otro si es un superconjunto: es decir, contiene todas las llaves de la otra. Por ejemplo, ((a) (b)) or ((un B)) es mejor que ((a)). El único usuario de esta rutina es grouping_planner().

- **pathkeys_contained_in**

Caso especial de compare_pathkeys: apenas desea saber si keys2 esta por lo menos ordenado como keys1.

- **noncanonical_pathkeys_contained_in**

Igual que la rutina anterior, pero se llama cuando no se tiene pathkeys canónicos.

- **get_cheapest_path_for_pathkeys**

Encuentre el path más barato (según el criterio especificado) que satisfacen los pathkeys dados. 'path' es una lista de paths posibles que generan la misma relación. 'pathkeys representan un orden (sort) requerido (ya canonizado) y 'cost_criterion' es STARTUP_COST o TOTAL_COST.

- **get_cheapest_fractional_path_for_pathkeys**

Encuentre el path más barato (para recuperar una fracción especificada de todos las tuplas) que satisface los pathkeys dados. En compare_fractional_path_costs() se encuentra la interpretación del parámetro fracción. 'paths' son una lista de trayectorias posibles que generan la misma relación. 'pathkeys' representan un orden (sort) requerido (ya canonizado) y 'fraction' es la fracción de las tuplas totales esperados para ser recuperadas.

- **build_index_pathkeys**

Construye un pathkeys list, que describe el orden (sort) indicado por un índex scan usado con el índice dado.

- **find_indexkey_var**

Encuentra o hace un nodo Var para el atributo especificado del rel. Primero busca el var en target list del rel, porque es fácil y rápido. Pero si el var no pudo estar allí, atrapa un nodo Var de la manera dura.

- **build_join_pathkeys**

Construye las path keys para una join_relation construida por el mergejoin o nestloop join. Estas keys deben incluir todos las key vars de la outer path más cualquier vars de la inner path que sea equijoined a los outer vars.

- **make_pathkeys_for_sortclauses**

Genera el pathkeys list que representa el orden especificado por la lista de SortClauses (o GroupClauses). El resultado no está en forma canónica, sino se debe pasar a través de canonicalize_pathkeys() antes de que pueda ser utilizado para las comparaciones o las órdenes de etiquetado de orden (sort) de la relación.

- **cache_mergeclause_pathkeys**

Hace los pathkeys válidos almacenados en un mergeclause restrictinfo.

- **find_mergeclauses_for_pathkeys**

Esta rutina procura encontrar un conjunto de mergeclauses que pueden ser usados con un orden (sort) especificado para una de las relaciones de la entrada. 'pathkeys' es una pathkeys list que demuestra el orden de un path de entrada. No importa si ésta es para inner u outer path. 'restrictinfos' es una lista de las cláusulas de restricciones de un mergejoin para el join relation que es formado.

- **make_pathkeys_for_mergeclauses**

Construye un pathkey list representando el orden (sort) que se debe aplicar a un path para hacerlo utilizable por los mergeclauses dados. 'mergeclauses' es una lista de RestrictInfos para las cláusulas del mergejoin que serán utilizadas en un merge join. 'rel' es la relación a la que los pathkeys se aplicarán (inner o outer del propuesto join rel). Retorna un pathkeys list que se puede aplicar a la relación especificada.

- **pathkeys_useful_for_merging**

Cuenta el número de los pathkeys que pueden ser útiles para los mergejoins sobre la relación dada (mirando sus listas del joininfo).

- **pathkeys_useful_for_ordering**

Cuenta el número de pathkeys que son útiles para satisfacer el orden (sort) solicitado por el query.

- **truncate_useless_pathkeys**

Corta el pathkey list dado para solo usar pathkeys.

**Archivo tidpath.c**

Rutinas para determinar cuales tids son usables para escanear una relación dada, y crear TidPaths acordes.

Incluye las siguientes librerías:

postgres.h
math.h
catalog/pg_operator.h
optimizer/clauses.h
optimizer/cost.h
optimizer/pathnode.h
optimizer/paths.h
parser/parse_coerce.h
utils/lsyscache.h

Entre sus funciones se encuentran:

- **isEvaluable**

Revisa de que tipo es un varno recibido, puede ser Const, Param, Var o funcclause

- **TidequalClause**

Revisa los campos de una estructura nodo para ver si tiene una cláusula equal y sus argumentos.
El segundo parámetro podría ser un opclause, se extrae el nodo derecho si el opclause es CTID=... o el nodo izquierdo se es opclause es ...=CTID

- **TidqualFromExpr**

Extrae la lista de valores CTID desde un nodo expr específico. Cuando el nodo expr es un or_clause, se trata de extraer los valores de CTID desde todos los nodos miembros. Sin embargo se podría descartarlos todos si no se pudiera extraer los valores CTID desde un nodo miembro. Cuando el nodo expr es un and_clause, se retorna la lista de valores CTID si se puede extraer los valores CTID desde un nodo miembro.

- **TidqualFromRestrictinfo**

- **create_tidscan_joinpaths**

Crea los paths de innerjoin si hay cláusulas join convenientes.

- **create_tidscan_paths**

Crea los paths correspondientes a escaneos directos de tid de la relación dada. Los paths candidatos son adheridos al pathlist de la relación (utiliza ad_path).

**ANALISIS GEQO**

Implementación de algoritmos genéticos para el optimizador.
/geqo es el planer "optimización genética" separado. Este hace una búsqueda semi-randomica a través del espacio del árbol join, en lugar de considerar exaustivamente todos los posibles árboles join. (Pero cada join considerado por /geqo es dado a /path para crear rutas para ellos, así que consideramos todos las posibles implementaciones de rutas para cada pareja de join especifica aun en el modo GEQO.)

En particular el funcionamiento de los algoritmos de este directorio son bastante diferentes del resto de rutinas del optimizador y de backend en general.
De hecho están basados en algoritmos ya implementados (mejor ruta, mutación y piscina de genes); es demasiado irregular y especializado para explicarlo someramente.

\postgresql-7.3.4\src\backend\optimizer\geqo

geqo_copy.c
geco_cx.c
geco_erx.c
geco_eval.c
geco_main.c
geco_misc.c
geco_mutation.c
geco_ox1.c
geco_ox2.c
geco_pmx.c
geco_pool.c
geco_px.c
geco_recombination.c
geco_selection.c

\postgresql-7.3.4\src\include\optimizer
Los mismos archivos *.h mas otros mencionados.

**ANALISIS PLAN**

**Archivo createplan.c**

Rutinas para crear el plan deseado para el procesamiento de una consulta. El planning esta completo, solo se necesita convertir el Path (Ruta) seleccionado en un Plan.

Existen funciones static de cada tipo para cada plan posible: (scan, seqscan, tidscan, function,... hashjoin)

- **create_plan**

Inicia aquí la transformación. Crea el plan de acceso para una consulta mediante el trazado hacia atrás a través de la cadena deseada de pathnodes, empezando en el nodo 'best_path'. Por cada pathnode encontrado:
1. Crea el correspondiente plan node conteniendo su apropiado id, target list, e información cualificativa.
2. Modifica cláusulas qual de los nodos join de modo que atributos de subplan son referenciados usando valores relativos.
3. Target lists no son modificados, pero lo serán en setref.c.

best_path es la mejor ruta de acceso y retorna un Plan tree.

- **create_scan_plan**

Invocada por la principal create_plan(). Crea un plan scan para la relación padre de 'best_path'. Retorna un Plan node.

Extrae las restriction clauses relevantes desde la relación padre; el executor debe aplicar todas estas restricciones durante el scan.

- **create_join_plan**

Invocada por la principal create_plan(). Crea un join plan para 'best_path' y (recursivamente) planes para sus inner y outer paths. Retorna un Plan node.

- **create_append_plan**

Invocada por la principal create_plan(). Crea un Append plan para 'best_path' y (recursivamente) plans para sus subpaths. Retorna un Plan node.

- **METODOS PARA SCAN BASE-RELATION**

- **create_seqscan_plan**

Retorna un seqscan plan para la base relation scanned por 'best_path' con restriccion clauses 'scan_clauses' y targetlist 'tlist'.

- **create_indexscan_plan**

Retorna un plan indexscan para la relación base escaneada por 'best_path' con cláusulas de restricción 'scan_clauses' y targetlist 'tlist'. Finalmente esta listo para construir el plan node. Retorna scan_plan.

- **create_tidscan_plan**

Retorna un tidscan plan para la relación base escaneada por 'best_path' con cláusulas de restricción 'scan_clauses' y targetlist 'tlist'.

- **create_subqueryscan_plan**

Retorna un subqueryscan plan para la relación base escaneada por 'best_path' con cláusulas de restricción 'scan_clauses' y targetlist 'tlist'.

- **create_functionscan_plan**

Retorna functionscan plan para la relación base escaneada por 'best_path' con cláusulas de restricción 'scan_clauses' y targetlist 'tlist'.

- **METODOS JOIN**

Una nota general sobre el procesamiento de join_references() en estas rutinas:
Una vez se ha cambiado un nodo Var para referirse a un subplan output, en vez de a la relación original, este ya no será mas equal() a un nodo Var no modificado para la misma var. Así, no se puede fácilmente comparar cláusulas qual de referencia adjustada con cláusulas que no han sido ajustadas. Afortunadamente, eso no parece ser necesario; todas las decisiones son hechas antes de que se haga los ajustes de referencias.

Una solución limpia sería no llamar aquí a join_references(), y dejarlo para que setrefs.c lo haga al final de la construcción del plan tree. Pero eso hará un switch_outer() mucho más complicado, y necesita más cuidado para conseguir que setrefs.c haga lo correcto con nestloop inner indexscan quals. Así, se hacen aquí únicamente los ajustes de referencia al subplan para quals de nodos join.

- **create_nestloop_plan**
- **create_mergejoin_plan**
- **create_hashjoin_plan**

- **RUTINAS DE SOPORTE**

- **fix_indxqual_references**

Ajusta cláusulas indexqual a la forma del indexqual que requiere la maquinaria del ejecutor.

- **fix_indxqual_sublist**

Ajusta la sublista de indexquals para un scan particular para cada qual clause, conmuta si es necesario para poner el indexkey operand a la izq, y entonces ajusta sus varattno (no hay necesidad de cambiar al otro lado de la clause).
También cambia el operador si se necesita, y chequea comportamiento de index perdidos.
Retorna dos listas: la lista de los indexquals ajustados, y la lista (usual/vacía) de las cláusulas originales que deben ser re-chequeadas como qpquals porque el index está confundido para este operator type.

- **fix_indxqual_operand**

- **switch_outer**

Dado una lista de merge o hash joinclauses, reordena los elementos dentro de las cláusulas, así la outer join variable esta sobre la izq y la inner esta sobre la derecha. La lista original no es tocada; una lista modificada es retornada.

- **order_qual_clauses**

Ordena la lista de clauses según criterios como se quiera chequearlas en tiempo de ejecución. El orden está dado por costo ejecución/selectividad.

- **copy_path_costsize**

Copia información de costo y tamaño desde el nodo Path al nodo Plan creado de él. El executor no utilizará esta información, pero es necesaria para EXPLAIN.

- **copy_plan_costsize**

Copia información de costo y tamaño desde un nodo plan menor hacia un nodo insertado.

- **RUTINAS DE CONSTRUCCION DEL PLAN NODE**

Algunas de estas son exportadas porque ellas son llamadas para construir nodos plan en contextos donde no se esta derivando el plan node desde un path node.

En general se crea un nodo del tipo de plan que se va a crear (seqscan, indexscan, subquery, hashjoin, etc) con sus particularidades, y un nodo Plan que contiene los elementos comunes a cualquier plan. Se llenan los valores obtenidos hasta ahora en las listas respectivas.

- **make_seqscan**
- **make_indexscan**
- **make_tidscan**
- **make_subqueryscan**
- **make_functionscan**

- **make_append**

Inicializa los costos a 0, recorre subnodos actualizando valores.

- **make_nestloop**
- **make_hashjoin**
- **make_hash**
- **make_mergejoin**

- **make_sort**

Para usar make_sort directamente, se debe ya tener marcado la lista con reskey y reskeyop info. Las mejor keys que no sean redundantes, (ie, mejor que hubiera tlist ítems marcados con cada key number desde 1 hast keycount), o el executor se confundirá.

- **make_sort_from_pathkeys**

Crea sort plan para ordenar de acuerdo a las pathkeys dadas.
'tlist' es el target list del input plan.
'lefttree' es el nodo el cual produce input tuples.
'pathkeys' es la lista de pathkeys por las cuales el resultado será ordenado.

Se debe convertir la información pathkey en campos reskey y reskeyop de nodos resdom en el target list del sort plan.

- **make_material**

Hace startup y total costs igual a total cost de input plan; solo afecta EXPLAIN.

- FUNCIONES PARA USO EXTERNO

- **make_agg**

Hace cálculos. Carga un cpu_operator_cost por función agregada, por tupla de entrada y estima el número de grupos para aplicar las agregadas.

- **make_group**

Carga un cpu_operator_cost por comparación por tupla de entrada. Se asume que todas las columnas son comparadas en la mayoría de las tuplas.

- **make_unique**

distinctList es una list de SortClauses, identificando los ítems targetlist que deberían ser considerados por el filtro Unique.

- **make_setop**

distinctList es una lista de SortClauses, identificando los ítems targetlist que deberían ser considerados por el filtro SetOp.

- **make_limit**
- **make_result**

**Archivo planmain.c**

Rutinas de plan para consultas simples.
El punto de entrada del nivel superior del planner/optimizar esta en planner.c. Este es el código principal (main) para planear operaciones básicas de join, subselects, herencia, agregadas, agrupamiento.

Incluye las siguientes librerías:

postgres.h
optimizer/clauses.h
optimizer/cost.h
optimizer/pathnode.h
optimizer/paths.h
optimizer/planmain.h

Entre sus funciones se encuentran:

- **query_planner**

Genera un path para una consulta básica, la cual podría involucrar joins.

Puesto que query_planner no maneja el procesamiento en el nivel superior (toplevel) (agrupamiento, ordenamiento, etc.) no puede seleccionar el mejor path por si mismo. Este selecciona dos paths:
la ruta más barata que produce todas las tuplas requeridas, independientemente de alguna consideración de ordenamiento, y la ruta más barata que produce fracciones de tuplas requeridas en el requerido ordenamiento. El llamado (grouping_planner) daría la decisión final de cual usar.

Parámetros de entrada:

| | |
|---|---|
| root | es la consulta al plan |
| tlist | es la lista target a la consulta que debería producir. |
| tuple_fraction | es la fracción de tupla que serian recibidas. |

Parámetros de salida:

| | |
|---|---|
| *cheapest_path | recibe la ruta global más barata para la consulta. |
| *sorted_path | recibe la ruta preordenada más barata para la consulta. |

Nota: El nodo Query también incluye un campo query_pathkeys, el cual es una entrada y una salida del query_planner(). El valor de entrada señala que el indicador de ordenamiento es deseado en la salida final del plan. Pero este valor aun no está en "forma canónica", puesto que necesita que la tarea se haga (low_level indxpath.c).

tuple_fraction es interpretado como:

| | |
|---|---|
| 0: | todas las tuplas que serán recibidas(caso normal) |
| 0 < tuple_fraction < 1: | da la fracción de tuplas disponibles del plan que serán recibidas. |
| tuple_fraction >=1: | tuple_fraction es el número absoluto de tuplas que serán recibidas (Limite). |

**Archivo initsplan.c**

Target list, qualification, joininfo rutinas de inicialización

Incluye las siguientes librerías:

postgres.h
catalog/pg_operator.h
catalog/pg_type.h
nodes/makefuncs.h
optimizer/clauses.h
optimizer/cost.h
optimizer/joininfo.h
optimizer/pathnode.h
optimizer/paths.h
optimizer/planmain.h
optimizer/tlist.h
optimizer/var.h
parser/parsetree.h
parser/parse_expr.h
parser/parse_oper.h
utils/builtins.h
utils/lsyscache.h
utils/syscache.h

Entre sus funciones se encuentran:

- **JOIN TREES**

- **add_base_rels_to_query**

Escanea los jointree de las consultas y crea baserel RelOptInfos para todas las relaciones base (ie, tablas, subquery, y funciones RTEs) que aparecen en el jointree.

Al final de este proceso, resultaría un baserel RelOptInfo para todos non-join RTE usados en la consulta. Por eso, esta rutina es la única que debería llamar a build_base_rel. Pero build_other_rel será usada luego para estructurar relaciones por herencia.

- **TARGET LISTS**

- **build_base_rel_tlists**

Agrega entradas targetlist para cada var necesitando en el final tlist de las consultas las relaciones base adecuadas.

- **add_vars_to_targetlist**

Por cada variable que aparece en la lista, se agrega en la propia targetlist de la relación si ya no esta presente, y marca la variable como existente para indicar join.

- **QUALIFICATIONS**

- **distribute_quals_to_rels**

Escanea recursivamente los join tree de las consultas para WHERE y cláusulas JOIN/ON, y los agrega a las listas RestrictInfo y JoinInfo apropiadas posesiones del base RelOptInfos. También, base RelOptInfos son marcados con información outerjoinset, que ayudan en el posicionamiento de la condiciones de las cláusulas que aparecen por encima outer joins.

NOTA: cuando se está tratando con inner joins, es apropiado permitir a la cláusula qual ser evaluado en el nivel más bajo donde todas las variables mencionadas están disponibles.

- **mark_baserels_for_outer_join**

Marca todas las relaciones base listadas en 'rels' como having the given outerjoinset.

- **distribute_qual_to_rels**

Agrega información a uno de los campos 'RestrictInfo' o 'JoinInfo' (dependiendo si la cláusula es un join) de cada relación base mencionada en la cláusula. Un nodo RestrictInfo es creado y agregado a la lista apropiada de cada rel. También, si la cláusula usa un operador mergejoinable y no retrasa reglas por outer-join, entre a las expresiones del lado izquierdo y derecho dentro de las listas de la consulta de las vars equijoined.

'clause' la condición de la cláusula será distribuida.
'ispusheddown' si TRUE, obliga a la cláusula a ser marcada 'ispusheddown'
(esto indica que la cláusula vendría de un FromExpr, no de un JoinExpr)
'isdeduced' TRUE si la condición vendría de una deducción implied-equality.
'outerjoin_nonnullable': NULL si no es una condición outer-join, sino el conjunto de
baserels, aparecerán en el lado (nonnullable) externo del join.
'qualscope' conjunto de baserels; el abarca el alcance sintáctico de las condiciones

'qualscope' identifica de qué nivel JOIN la condición es llamada. Para una condición de nivel superior (condición WHERE), qualscope lista todos los ids baserel y en adición 'ispusheddown' será TRUE.

- **process_implied_equality**

Chequea hasta mirar si ya se tiene un ítem restrictinfo que dice item1 = item2, y crea uno si no; o si delete_it es true, remueve alguno semejante al ítem restrictinfo.

Este proceso es consecuencia de la transitividad de la igualdad mergejoin:

si nosotros tenemos una cláusula mergejoinable A = B y B = C, nosotros podemos deducir A = C (donde = es un operador mergejoinable apropiado). Mire path/pathkeys.c para más detalles.

- **qual_is_redundant**

Detecta si una condición implied-equality que retorna una cláusula de restricción para relación base simple es redundante con una cláusula de restricción ya conocida de la relación. Esto ocurre con, por ejemplo,

```
SELECT * FROM tab WHERE f1 = f2 AND f2 = f3;
```

Se necesita suprimir la condición redundante para evitar computar too-small selectividad, no mencionar el gasto en el tiempo de ejecución.

Note: condiciones de la forma "var = const" no son consideradas redundantes, únicamente estas de la forma "var = var". Esto es necesario porque cuando se tenga constantes en un grupo implied-equality set, se use estrategias diferentes que suprimen toda deducción "var = var". Se debe por eso guardar todas las condiciones "var = const".

- **CHECKS FOR MERGEJOINABLE AND HASHJOINABLE CLAUSES**

- **check_mergejoinable**

Si las cláusulas restrictinfo's es mergejoinable, fija los campos info del mergejoin en el restrictinfo.

- **check_hashjoinable**

Si la cláusula restrictinfo's es hashjoinable, fija los campos info del hashjoin en el restrictinfo.

**Archivo planner.c**

Interfaz externa para optimizar el query.

Incluye las siguientes librerías:

postgres.h
catalog/pg_type.h
nodes/makefuncs.h
nodes/print.h
optimizer/clauses.h
optimizer/paths.h
optimizer/planmain.h
optimizer/planner.h
optimizer/prep.h
optimizer/subselect.h
optimizer/tlist.h
optimizer/var.h
parser/analyze.h
parser/parsetree.h
parser/parse_expr.h
rewrite/rewriteManip.h
utils/lsyscache.h

Entre sus funciones se encuentran:

- **planner**

Función de entrada al Optimizador que es llamada recursivamente. Activa el resto de las funciones de este archivo.

- **subquery_planner**

Invoca al planner en un subquery. Recurre aquí por cada sub-SELECT encontrado en el query tree.

Parse es el querytree producido por el parser y el rewriter. tuple_fraction es la fracción de tuplas que serán recuperadas. tuple_fraction se interpreta según lo explicado por el grouping_planner.

Básicamente, esta rutina hace las cosas que deben ser hechas solamente una vez por objeto Query. Entonces llama el grouping_planner. El grouping_planner se podrá invocar recursivamente en el mismo objeto Query; subquery_planner será llamado recursivamente para manejar nodos sub-Query encontrados dentro de las expresiones del Query y rangetable. Retorna un Query plan.

- **pull_up_subqueries**

Busca subqueries en el rangetable las cuales pueden ser subidas a la query padre.
Si el subquery no tiene ninguna característica especial como grouping/aggregation entonces lo combina en el jointree del padre.

Un aspecto difícil de este código es que si sube un subquery tiene que sustituir los Vars que refieren las salidas del subquery a través del query padre, incluyendo los quals unidos a los nodos jointree que se están procesando actualmente.

- **preprocess_jointree**

Intenta simplificar el jointree de un Query. Si tiene éxito subiendo un subquery entonces puede ser que forme un jointree en el cual un FromExpr es hijo directo de otro FromExpr. En ese caso se puede considerar el combinar los dos FromExprs en uno. Ésta es una conversión opcional, puesto que el planificador trabajará correctamente de cualquier manera. Pero puede encontrar un plan mejor (costando más tiempo del planeamiento) si combina los dos nodos.

- **preprocess_expression**

Hace el trabajo del proceso previo del subquery_planner para una expresión, que pueda ser un targetlist, o un WHERE clause (JOIN/ON incluyendo condiciones), o HAVING clause.

- **preprocess_qual_conditions**

Escanea recursivamente el jointree del Query y hace el trabajo del proceso previo del subquery_planner sobre cada condición qual encontrada en él.

- **inheritance_planner**

Genera un plan en el caso donde la relación resultante es una herencia fijada. Maneja este caso diferente de los casos donde una relación fuente es una herencia fijada. La herencia de la fuente se amplía en el fondo del PlanTree (como indica allpaths.c), solamente herencia del target tiene que ser ampliada en el top, porque para UPDATE, cada una de la relaciones necesita un targetlist diferente que empareja su propia columna.

- **grouping_planner**

Realiza los pasos del planeamiento relacionados con agrupar, las funciones agregadas, etc. Esto significa agregar procesos a nivel superior para el query plan producido por el query_planner.

parse es el querytree producido por el parser y rewriter. tuple_fraction es la fracción de tuplas que esperamos, la interpretamos como sigue:
< 0: determina la fracción por la inspección del query (caso normal).
0: espera que todos las tuplas sean recuperados
0 < tuple_fraction < 1: cuenta la fracción dada de tuplas disponible del plan que se recuperará.
> = 1: tuple_fraction es el número absoluto de tuplas que esperaba ser recuperado (por ejemplo, en una especificación de LIMIT).

El caso normal es pasar -1, pero algunos pasan valores > = 0 para eliminar la determinación de esta rutina de la fracción apropiada.

- **make_subplanTargetList**

Genera el target list apropiado cuando se requiere agrupar. Cuando grouping_planner inserta Aggregate y/o Group nodes arriba del resultado del query_planner, se debe pasar un target list diferente al query_planner que los nodos outer deben tener, generando así el target list correcto para el subplan.

El target list inicial pasada del parser contiene ya entradas para toda las expresiones ORDEN BY y GROUP BY, pero no tendrá entradas para las variables usadas solamente en HAVING clauses; es necesario agregar éstas variables al target list subplan. También, si estamos haciendo grouping o aggregation, aplanamos todas las expresiones excepto ítems del GRUPO BY dentro de sus variables componentes; las otras expresiones serán computadas por los nodos insertados por el subplan. Por ejemplo, dado un query como:
SELECT a+b, SUM(c+d) FROM table GROUP BY a+b; deseamos pasar este targetlist al subplan: a, b, c, d, a+b donde a+b será utilizada por los pasos de Sort/Group, y otros targets serán utilizados para computar los resultados finales (en el ejemplo podríamos suprimir teóricamente las blancos de a y de b y pasar abajo solamente c, d, a+b, pero esto no es realmente eliminar las referencias simples del var del subplan). Evitaremos hacer el cómputo adicional con a+b en el nivel externo; de esto se encargará replace_vars_with_subplan_refs() en setrefs.c.

- **make_groupplan**

Agrega un nodo group para el proceso GRUPO BY.

- **make_sortplan**

Agrega un nodo sort para implementar el proceso ORDER BY.

- **postprocess_setop_tlist**

Necesitamos transportar la clave de ordenación del orig_tlist a new_tlist.

**Archivo setrefs.c**

Hace el post procesamiento de un plan tree completo: fija referencias a las variables del subplan, y computa valores del regproc para los operadores.

Incluye las siguientes librerías:

nodes/makefuncs.h
nodes/nodeFuncs.h
optimizer/clauses.h
optimizer/planmain.h
optimizer/tlist.h
optimizer/var.h
parser/parsetree.h

Define las siguientes estructuras:
join_references_context
replace_vars_with_subplan_refs_context

Entre sus funciones se encuentran:

- **set_plan_references**

Este es el paso final del procesamiento del planner/optimizer. El plan tree esta completo; solo se debe ajustar algunos detalles representativos para conveniencia del executor. Se actualiza Vars en los nodos mas altos del plan para referirse a las de sus subplanes, y se computa regproc OIDs para operadores (ie, se busca la función que implementa cada op).
Se debe también construir listas de todos los nodos del subplan presentes en cada árbol de expresión de los nodos del plan.
set_plan_references atraviesa recursivamente todo el plan tree.

No retorna nada de interés, pero modifica campos internos de los nodos

- **fix_expr_references**

Hace la limpieza final en expresiones (targetlists o quals).
Esto consiste en ubicar la información del operador opcode para nodos Oper y adicionar subplanes a las listas de contenido de subplnes del Plan node.

- **set_join_references**

Modifica el target list de un nodo join para referenciar sus subplanes, colocando los varnos al OUTER o INER y colocando valores attno al número de domino de resultado de cualquier ítem de tupla de outer o inner join que corresponda.

Nota: Esta misma transformación ha sido aplicada a los quals de los join por el createplan.c. Este es un pequeño caso a realizar aquí para los targetlist y allá para los quals,

pero es más fácil. (Mire switch_outer() y el manejo de nestloop inner indexscans para ver por que)

Porque los quals son referenciados-ajustados mas pronto, no se puede hacer las comparaciones de equal() en medio de los qual y tlist var nodes durante el tiempo entre la creación del plan node por el createplan.c y su corrección por este módulo.

'join' es un join plan node
'rtable' es la range table asociada

- **set_uppernode_references**

Actualiza el targetlist y quals de un nodo del plan de nivel más alto para referir a las tuplas retornadas por su subplan del árbol izquierdo (lefttree).

Esta es usada por tipos de plan de entrada simple como Agg, Group, Result.

En la mayoría de los casos, se debe igualar los Vars individuales en la tlist y expresiones qual con elementos de la tlist del subplan (la que fue generada por flatten_tlist() desde esas mismas expresiones, entonces podría tener las variables requeridas). Hay una excepción importante, sin embargo: las expresiones Group By y Order By deben ser puestas dentro de las (unflattened) tlist del subplan. Si esas variables también son necesitadas en la salida entonces se desea referenciar los mejores elementos del tlist del subplan que recomputen la expresión.

- **join_references**

Crea un nuevo grupo de entradas de targetlist o cláusulas join qual cambiando los valores de varno/varattno de variables en las cláusulas para referirse a los valores de target list desde los target list de las relaciones de outer o inner join. También algunas variables de alias del join en las cláusulas son expandidas dentro de referencias a sus variables componentes.

Esto es usado en dos escenarios diferentes: una cláusula de join normal, donde todos los Vars en la cláusula deben ser remplazados por referencias OUTER o INNER; y un indexscan siendo usado sobre el lado inner de un join nestloop. En el último caso se desea remplazar las Vars de la relación outer por referencias OUTER, pero no se toca las Vars de las relaciones inner.

Para un join normal, acceptable_rel sería cero de modo que cualquier fallo para igualar un Var será reportado como un error. Para el caso de indexscan, pasa inner_tlist = NIL y acceptable_rel = a el ID de la relación inner.

'clauses' es el targetlist o lista de cláusulas join
'rtable' es al range table comun
'outer_tlist' es la target list de la relacion del outer join
'inner_tlist' es la target list de la relación del inner join o NIL

'acceptable_rel' es cero o el índice del rangetable de la relación cuyas Vars aparecen en la cláusula sin provocar algún error

Retorna el nuevo árbol de expresión. La estructura de cláusula original no es modificada.

- **join_references_mutator**

Busca los var en las tlists de entrada

- **replace_vars_with_subplan_refs**

Esta rutina modifica un árbol de expresión de modo que todos los Var nodes referencien target nodes de un subplan. Esto es usado para componer los target y expresiones qual de plan nodes de más alto nivel de no-joins.

Un error es levantado si una var no igualada puede ser encontrada en la tlist del subplan, entonces esta rutina solo sería aplicada a nodos cuyas targetlist de subplan eran generadas via flatten_tlist() o algún método parecido.

Si tlist_has_non_vars es verdad, entonces tratamos de igualar todas las expresiones contra elementos de la tlist del subplan, de modo que podamos evitar recomputar expresiones que fueron ya computadas por el subplan (Esto es relativamente costoso, entonces no deseamos intentarlo en los casos comunes donde la tlist del subplan es solo una lista flattened de Vars).

'node': el árbol a ser corregido (un target ítem o qual)
'subvarno': varno a ser asignada a todos los Vars.
'subplan_targetlist': target list para el subplan.
'tlist_has_non_vars': verdad si el subplan_targetlist contiene expresiones non-Var.

El árbol resultante es una copia del original en el cual todos los nodos var tienen varno = subvarno, varattno = resno de correspondencia al target del subplan. El árbol original no es modificado.

- **replace_vars_with_subplan_refs_mutator**

También trata de igualar más expresiones complicadas, si el tlist tiene alguna.

- **fix_opids**

Calcula el campo opid desde opno para cada nodo Oper en el árbol dado. El árbol dado puede ser cualquier expression_tree_walker manejada.
El argumento es modificado in-place (Esto esta bien desde que deseemos el mismo cambio para cualquier nodo, aun cuando este consiga visitar más de una vez directamente a la estructura compartida).

- **fix_opids_walker**

**Archivo subselect.c**

Planea las rutinas para los subselects y parámetros

Incluye las siguientes librerías:

catalog/pg_operator.h
catalog/pg_type.h
nodes/makefuncs.h
optimizer/clauses.h
optimizer/cost.h
optimizer/planmain.h
optimizer/planner.h
optimizer/subselect.h
parser/parsetree.h
parser/parse_expr.h
parser/parse_oper.h
utils/syscache.h

Define las sig estructuras:
finalize_primnode_results;

Entre sus funciones se encuentran:

- **new_param**

Crea una nueva entrada en la lista PlannerParamVar, y retorna su índice var contiene los datos a ser copiados, excepto para varlevelsup que es dado desde el valor del nivel absoluto dado por varlevel

- **replace_var**

Genera un nodo Param para remplazar el Var dado, quien se espera que tenga varlevelsup >0 (ie, este no es local).
Si hay una entrada PlannerParamVar lista par esta misma Var, solo es usada.
Nota: En querytrees suficientemente complicados, esto es posible para el mismo varno/varlevel para referir a los diferentes RTEs en diferentes partes del parsetree, entonces esos diferentes campos podrían terminar compartiendo los mismos números Param. Tanto como se chequea los vartype como buenos, se cree que esta especie de aliasing no causara problemas. El campo correcto se guardaría dentro del slot Param en ejecución en cada parte del árbol.

- **make_subplan**

Convierte un Sublink descubierto (como creado por el parser) en un Subplan.
Aquí se hace parte del cálculo de costos.

- **finalize_primnode**

Construye listas de subplanes y parámetros apareciendo en el árbol de expresión dado.
Nota: Los ítems son adheridos a listas aprobadas, los llamados deben inicializar las listas en NIL antes del primer llamado.
Nota: La lista del subplan que es construida aquí y asignada al campo subplan del plan será remplazada con una lista up-to-date en set_plan_references().

- **SS_replace_correlation_vars** ()

Reemplaza los vars de correlación (vars de nivel superior) con Parámetros

- **replace_correlation_vars_mutator** ()

- **SS_process_sublinks**

Expande los Sublinks a Subplanes en la expresión dada

- **process_sublinks_mutator**

- **SS_finalize_plan**

**ANALISIS PREP**

**Archivo prepqual.c**

Rutinas para preprocesar expresiones y cualificaciones. Estas rutinas convierten una expresión booleana arbitraria en forma normal conjuntiva (CNF) o forma normal disyuntiva (DNF). La normalización se realiza solamente en la porción superior AND/OR/NOT del árbol dado; no procura normalizar las expresiones booleanas que puede aparecer como argumentos de operadores o de funciones en el árbol.

Las calificaciones del Query (WHERE clauses) se transforman ordinariamente en CNF, es decir formas AND-of-ORs, porque el optimizador puede utilizar cualquier cláusula AND independiente calificación de filtración. Sin embargo, los quals que se expresan naturalmente como OR-of-ANDs sufren un crecimiento exponencial de tamaño en esta transformación, así que también considera convertir a DNF (OR-of-ANDs), y también puede solo dejarlas si ambas transformaciones causan un crecimiento no razonable. La forma OR-of-ANDs es útil para implementar indexscan, así que se elige este formato cuando hay solo una relación implicada.

canonicalize_qual() hace una conversión "inteligente" para CNF o a DNF, por las consideraciones antedichas, mientras que cnfify() y dnfify() realizan simplemente la transformación exigida. Los dos últimos pueden ser código muerto eventual.

El parser entrega AND y OR como operadores binarios puros, así un qual como (A = 1) OR (A = 2) OR (A = 3)...producirá un nested parsetree (OR (A = 1) (OR (A = 2) (OR (A = 3)...))). En realidad, el optimizador y el ejecutor miran AND y OR como operadores de n-argumentos, así que este árbol puede ser aplanado a (OR (A = 1) (A = 2) (A = 3)...) que es la responsabilidad de las rutinas como flatten_andors() hace la transformación básica sin asunciones iniciales. pull_ands() y pull_ors() se utilizan para mantener aplanados los AND/OR del árbol después de que las transformaciones locales pudieron introducir nested AND/ORs.

Entre sus funciones se encuentran:

- **canonicalize_qual**

Convierte una calificación a la forma normalizada más útil. Si 'removeAndFlag' es true entonces él quita el AND explícito del nivel superior, produciendo una lista de condiciones implicitly-ANDed. Si no, una expresión booleana regular se retorna. Puesto que la mayoría de los llamadores pasan 'true', se prefiere declarar el resultado como List *, no Expr *. Intenta calcular si hay AND/OR selectivities mirando los índices para soportar una implementación de indexscan de un DNF qual. Podría incluso intentar convertir las cláusulas CNF que mencionan una sola relación en una simple cláusula DNF para ver si resulta más baratas esa implementación.

- **cnfify**

Convierte una calificación a la forma normal conjuntiva aplicando normalizaciones sucesivas.

- **dnfify**

Convierte una calificación a la forma normal disyuntiva aplicando normalizaciones sucesivas.

- **flatten_andors**

Dada una calificación, simplifica cláusulas nested AND/OR en cláusulas planas AND/OR con más argumentos.

- **pull_ors**

Empuja los argumentos de un cláusula nested OR dentro de otra cláusula OR superior en la lista de argumentos del padre.

- **pull_ands**

Empuja los argumentos de una cláusula nested AND dentro de otra cláusula AND superior en la lista de argumentos del padre.

- **find_nots**

Recorre la calificación, buscando 'NOT's para tener cuidado. Para cláusulas NOT, aplica push_not() para intentar empujar hacia abajo el 'NOT'. Para todos los otros tipos de cláusulas, simplemente se hace recursivo.

- **push_nots**

Empuja hacia abajo 'NOT' lo más lejos posible. La entrada es una expresión que se negará.

- **find_ors**

Dado un árbol de la calificación con el 'NOT' empujado hacia abajo, lo convierte a un árbol en CNF en varias ocasiones aplicando la regla:
("OR" A ("AND" B C)) => ("AND" ("OR" A B) ("OR" A C)). Las cláusulas 'OR' serán retornadas siempre en cláusulas 'AND' si contienen cualesquier subclauses 'AND'.

- **or_normalize**

Dada una lista de exprs que son 'OR'ed junto, intenta aplicar la ley distributibutiva:
("OR" A ("AND" B C)) => ("AND" ("OR" A B) ("OR" A C)) para convertir la cláusula OR de nivel superior a una cláusula AND de nivel superior.

- **find_ands**

Dado un árbol de la calificación con el 'NOT' empujado hacia abajo, lo convierte a un árbol en DNF en varias ocasiones aplicando la regla:
("AND" A ("OR" B C)) => ("OR" ("AND" A B) ("AND" A C)). Las cláusulas 'AND' serán retornadas siempre en cláusulas 'OR' si contienen cualesquier subclauses 'OR'.

- **and_normalize**

Dada una lista de exprs que son 'AND'ed junto, intenta aplicar la ley distributiva:
("AND" A ("OR" B C)) => ("OR" ("AND" A B) ("AND" A C)) para convertir la cláusula AND de nivel superior a una cláusula OR de nivel superior.

- **qual_cleanup**

Fija encima una calificación quitando las entradas duplicadas (que podrían ser creadas durante la normalización, si subexpressions idénticas de diferentes partes del árbol se reúnen). También, comprueba las cláusulas AND u OR con solamente una subexpression restante.

- **count_bool_nodes**

Ayuda a la heurística en canonicalize_qual(). Cuenta el número de nodos que son entradas para el nivel superior AND/OR/NOT de un árbol qual, y estiman cuántos nodos aparecerán en la expresión equivalente CNF o DNF. Esto es justo un cálculo aproximado; no detalla muy bien con NOT's, y por supuesto no puede detectar posibles simplificaciones para eliminar subclauses duplicadas. La idea es solo para determinar económicamente si CNF será marcado y no DNF o viceversa.

**Archivo preptlist.c**

Rutinas para preprocesar el target list del parse tree. Este módulo tiene cuidado de alterar el targetlist del query como sea necesario para queries INSERT, UPDATE, y DELETE. Para INSERT y UPDATE, el targetlist debe contener una entrada para cada atributo de la relación target en el orden correcto. Para ambos UPDATE y DELETE, necesita una entrada junk (chatarra) del targetlist esperando el atributo CTID. El ejecutor confía en esto para encontrar la tupla para ser reemplazada/borrada.

Entre sus funciones se encuentran:

- **preprocess_targetlist**

Driver para preprocesar el targetlist del parser tree. Retorna el nuevo targetlist

- **targetlist**

Dado un target list según lo generado por el parser y una relación resultado, agrega las entradas targetlist para cualquier atributo que falta, y asegura que atributos non-junk aparecen en el orden apropiado. Si se intentan poner más procesos aquí, se debe considerar que deben ir en el rewriteTargetList() del rewriter.

**Archivo prepunion.c**

Rutinas para planear set-operation queries. Hay también un código aquí para apoyar el planeamiento de las querys que utilizan la herencia.

Entre sus funciones se encuentran:

- **plan_set_operations**

Planea el query para un árbol de set operations (UNION/INTERSECT/EXCEPT). Esta rutina trata solamente el árbol setOperations del query dado. Cualquier ORDER BY solicitado en parser->sortClause será agregado cuando retorna al grouping_planner.

- **recurse_set_operations**

Maneja recursivamente un paso en un árbol de set operations.

- **generate_union_plan**

Genera un plan para un nodo UNION o UNION ALL.

- **generate_nonunion_plan**

Genera un plan para un nodo INTERSECT, INTERSECT ALL, EXCEPT, o EXCEPT ALL.

- **generate_setop_tlist**

Genera el targetlist para un nodo set-operation.

- **tlist_same_datatypes**

tlist tiene los mismos datatypes que los colTypes solicitados?

- **find_all_inheritors**

Retorna una lista entera de relids incluyendo el rel dado más todas las relaciones que heredan de él, directa o indirectamente.

- **expand_inherted_rtentry**

Comprueba si una entrada rangetable represente una herencia fijada. Si es así agrega las entradas para todas las tablas hijas al rangetable del query, y retorna una lista de enteros de los índices del RT para la herencia entera fijada (padre y los hijos). Si no, retorna NIL.

Cuando dup_parent es falso, el RT índice inicialmente dado es parte de la lista retornada. Cuando dup_parent es verdad, el RT índice dado no esta en la lista retornada; un RTE duplicado será hecho para la tabla padre.

Una tabla sin hijos nunca se considera una herencia fijada; por lo tanto el resultado nunca será una lista de un elemento. Será cualquier vacío o tendrá dos o más elementos. Después de que esta rutina se ejecute, el RTE especificado tendrá siempre su flag inh despejada,

hayan o no algunos hijos. Esto asegura no ampliará el mismo RTE dos veces, que ocurriría de otra manera para el caso de una relación heredada de target de UPDATE/DELETE.

- **adjust_inherited_attrs**

Copia la pregunta o la expresión especificada y traduce los Vars referenciando al old_rt_index para referir al new_rt_index. También ajusta varattno para emparejar la nueva tabla por nombre de la columna, y el número de la columna. Este corte hace posible para que las tablas hijo tengan diversas posiciones de columna por el "mismo" atributo padre, que ayuda ALTER TABLE ADD COLUMN. Desafortunadamente esto no es muy transparente; hay otros lugares donde empujan las cosas abajo si los hijos y los padres no tienen los mismos números de la columna para los atributos heredados (Mejor intentar arreglar ALTER TABLE).

- **adjust_inherited_tlist**

Ajusta las entradas del targetlist de una operación UPDATE heredada. Las expresiones ya han sido arregladas, pero es necesario cerciorarse de que el target resnos empareje la tabla hijo (no pueden, en el caso de una columna que fue agregada after-the-fact por ALTER TABLE). En algunos casos, esto puede forzar a reordenar el tlist para preservar ordenado el resno (todo este trabajo en casos especiales de modo que preptlist.c sea rápido para los casos típicos).

El tlist dado ya ha estado con expression_tree_mutator; por lo tanto los nodos TargetEntry son copias frescas que son aceptables. Pero los nodos Resdom no se han copiado; esto no es necesario para INSERT porque el INSERT no se puede heredar.

**ANALISIS UTIL**

**Archivo clauses.c**

Define rutinas para manipular las cláusulas de cualificación.

Incluye las siguientes librerías:

postgres.h
catalog/pg_operator.h
catalog/pg_type.h
catalog/pg_type.h
executor/executor.h
nodes/makefuncs.h
nodes/nodeFuncs.h
optimizer/clauses.h
optimizer/tlist.h
optimizer/var.h
parser/parsetree.h
utils/datum.h
utils/lsyscache.h
utils/syscache.h

Define las sig estructuras:

check_subplans_for_ungrouped_vars_context

Entre sus funciones se encuentran:

- **make_clause**

Se define la función de construcción del clause asignando los valores correspondientes al árbol Exp.

- **FUNCIONES DE LA CLÁUSULA OPERATOR**

- **is_opclause**

Retorna verdadero (t) si la cláusula es una cláusula operator: (op expr expr) o (op expr).
Nota Histórica: is_clause tuvo la misma funcionalidad, fue renombrada a is_opclause por claridad.

- **make_opclause**

Crea una cláusula dado el operando izquierdo y derecho del operador (si este no es nulo).

- **get_leftop**

Retorna el operando izquierdo de una cláusula de la forma (op expr expr) o (op expr).
Nota: Por razones históricas, el resultado es declarado Var *, aún cuando muchos llamadores pueden trabajar con resultados que no son Vars. El resultado real debería ser declarado Expr * o Node *.

- **get_rightop**

Retorna el operando derecho en una cláusula de la forma (op expr expr).
Nota: El resultado será nulo si es aplicado a una cláusula unaria

- **FUNCIONES DE LA CLÁUSULA FUNC**

- **is_funcclause**

Retorna verdadero (t) si la cláusula es una cláusula de función: (func { expr }).

- **make_funcclause**

Crea una cláusula de función dado el nodo FUNC y los argumentos funcionales.

- **FUNCIONES DE LA CLÁUSULA OR**

- **or_clause**

Retorna verdadero (t) si la cláusula es una cláusula 'or': (OR { expr }).

- **make_orclause**

Crea una cláusula 'or' dada una lista de las subclaúsulas

- **FUNCIONES DE LA CLÁUSULA NOT**

- **not_clause**

Retorna verdadero (t) si esta es una cláusula 'not': (NOT expr).

- **make_notclause**

Crea una cláusula 'not' dada la expresión a ser negada.

- **get_notclausearg**

Recobra o repara la cláusula con una cláusula 'not'

- **FUNCIONES DE LA CLÁUSULA AND**

- **and_clause**

Retorna verdadero (t) si los argumentos son una cláusula 'and': (AND { expr }).

- **make_andclause**

Crea una cláusula 'and' dado los argumentos en una lista

- **make_and_qual**

Variante de make_andclause para llevar al AND dos condiciones qual juntas. Las condiciones Qual tienen la propiedad que un nodetree nulo (NULL) es interpretado como verdadero (true).

- **make_ands_explicit & make_ands_implicit**

Algunas veces (tal como en el resultado de canonicalize_qual o la entrada de ExecQual), usamos listas de nodos expression con semánticas AND implícitas. Estas funciones convierten entre una lista de expresiones de semántica AND y la representación ordinaria de un expresión booleana. Note que una lista vacía es considera equivalente a verdadera (true).

- **MANIPULACIÓN DE LAS CLÁUSULAS DE FUNCIONES AGREGADAS**

- **contain_agg_clause**

Recursivamente busca nodos Aggref con una cláusula. Retorna verdadero si cualquier agregada es encontrada

- **contain_agg_clause_walker**

- **pull_agg_clause**

Recursivamente saca todos los nodos Aggref de un árbol de expresión. Retorna la lista de los nodos Aggref encontrados. Note que los nodos como si mismos no son copiados, solo referenciados

Nota: Esto también chequea por agregados nested, los que son un error.

- **pull_agg_clause_walker**

Aquí también se presentan funciones para:

- Soporte para conjuntos de retorno de expresiones.
- Manipulación de la cláusula de Subplan.
- Chequeo de cláusulas para funciones mutables.
- Chequeo de cláusulas para funciones volátiles.
- Chequeo para cláusulas "pseudo-constant".
- Pruebas en cláusulas de queries.
- Rutinas de manejo general de cláusulas.

**Archivo joininfo.c**

JoinInfo nodo de manipulación de rutinas.

Incluye las siguientes librerías:

postgres.h
optimizer/joininfo.h
optimizer/pathnode.h

Entre sus funciones se encuentran:

- **find_joininfo_node**

Encuentra el nodo del joininfo dentro de una entrada de la relación que corresponde a un join entre el ' this_rel ' y las relaciones en ' join_relids'. Si no hay tal nodo, retorna NULL. Al final la función retorna un nodo del joininfo, o NULL.

- **make_joininfo_node**

Encuentra el nodo del joininfo dentro de una entrada de la relación que corresponde a un join entre el ' this_rel ' y las relaciones en 'join_relids'. Un nuevo nodo se crea y se agrega al campo joininfo de la entrada de la relación si el deseado no puede ser encontrado. Al final la función retorna un nodo del joininfo.

- **add_join_clause_to_rels**

Para cada relación que participa en una cláusula del join, agrega el 'restrictinfo' a la lista apropiada del joininfo (creando una nueva lista y agregándolo al nodo rel apropiado en caso de necesidad).

'restrictinfo' describe la cláusula join.
'join_relids' es la lista de las relaciones que participan en la cláusula join (debe haber más de uno).

- **remove_join_clause_from_rels**

Borra 'restrictinfo' dentro de todas las listas del joininfo. Este invierte el efecto de add_join_clause_to_rels. Se utiliza cuando descubramos que una cláusula join es redundante.

'restrictinfo' describe la cláusula join.
'join_relids' son la lista de las relaciones que participan en la cláusula join (debe haber más de uno).

**Archivo pathnode.c**

Rutinas de manipulación de pathlists y crea nodos path.

Incluye las siguientes librerías:

postgres.h
math.h
catalog/pg_operator.h
executor/executor.h
miscadmin.h
nodes/plannodes.h
optimizer/cost.h
optimizer/pathnode.h
optimizer/paths.h
optimizer/restrictinfo.h
parser/parse_expr.h
parser/parse_oper.h
utils/memutils.h
utils/selfuncs.h
utils/syscache.h

Entre sus funciones se encuentran:

- **MISC. PATH UTILITIES**

- **compare_path_costs**

Retorna -1, 0, o +1 deacuerdo como path1 es el más barato, igual costo, o más caro que path2 de la especificación de criterios.

- **compare_path_fractional_costs**

Retorna -1, 0, o +1 deacuerdo como path1 es el más barato, igual costo, o más caro que path2 cogiendo la especificación de fracción del total de tuplas.

Si fracción es <= 0 o > 1, se interpreta esto como 1, ie, se selecciona el path con el total_cost más barato.

- **set_cheapest**

Halla los paths de mínimo costo de entre unas paths de relaciones, y los guarda en los campos cheapest-path (ruta más barata) de las relaciones.

Esto es normalmente llamando únicamente después de que se halla finalizado la construcción de la lista path del nodo rel.

Si se encuentra dos paths de idéntico costo, se trata de mantener el mejor, ordenado.

Los paths podrían tener sort ordenados sin relación, caso en el cual se puede únicamente adivinar cual podría ser el mejor para guardar, pero si uno es superior nosotros Definitivamente deberíamos guardarlo.

- **add_path**

Considere una implementación de path potencial para el rel especificado del padre, y agréguela al pathlist de los rel si es digno de consideración. Una path es digna si tiene o un mejor orden (pathkeys mejores) o un costo más barato (en cualquier dimensión) que cualesquiera de las viejas paths existentes.

A menos que sea parent_rel->pruneable falso, también se quita de los rel pathlist cualquier viejas paths que sean dominadas por new_path --- es decir, el new_path es más barato y por lo menos también bien ordenado.

El pathlist es guardado clasificado por la métrica TOTAL_COST, con paths más baratos en el frente. Ningún código depende de eso para la corrección; es simplemente un corte de la velocidad dentro de esta rutina. Hacerlo de esa manera hace más probablemente que se rechaze una path inferior después de algunas comparaciones, más bien que muchas comparaciones.

'parent_rel' es la entrada de la relación a la cual la path corresponde.
'new_path' es una trayectoria potencial para el parent_rel.

Retorna nada, pero modifican parent_rel->pathlist.

- **RUTINAS DE CREACIÓN DEL NODO PATH**

- **create_seqscan_path**

Crea una path que corresponde a una exploración secuencial, retornando el pathnode.

- **create_index_path**

Crea un nodo de trayectoria para una exploración del índice.

'rel' es el rel del padre.
'índice' es un índice en el 'rel'.
'restriction_clauses' es una lista de listas de los nodos RestrictInfo que se utilizará como condiciones qual del índice en la exploración.
'pathkeys' describe el ordenamiento del path.
'indexscandir' es ForwardScanDirection o BackwardScanDirection para un índice pedido, o NoMovementScanDirection para un índice desordenado.

La función en general retorna el nuevo nodo del path.

- **create_tidscan_path**

Crea una trayectoria que corresponde a una exploración del tid_direct, retornando el pathnode.

- **create_append_path**

Crea una trayectoria que corresponde a un plan Append, retornando el pathnode.

- **create_result_path**

Crea una trayectoria que corresponde a un plan Result, retornando el pathnode.

- **create_material_path**

Crea una trayectoria que corresponde a un plan Material, retornando el pathnode.

- **create_unique_path**

Crea una trayectoria que representa la eliminación de filas distintas de los datos de entrada.

Si es usada en todos, es probable ser llamada en varias ocasiones en el mismo rel; y el subpath de entrada siempre debe ser igual (la trayectoria cheapest_total para el rel). Así se deposita el resultado.

- **hash_safe_tlist** - pueden los datatypes del tlist dado ser hashed?

Se asume la agregación hashed que trabajará si el operador de igualdad de los datatype es marcada hashjoinable.
Mire también hash_safe_grouping en plan/planner.c.

- **create_subqueryscan_path**

Crea un path que corresponde a una exploración secuencial de un subquery, retorna el pathnode.

- **create_functionscan_path**

Crea un path que corresponde a una exploración secuencial de una función, retorna el pathnode.

- **create_nestloop_path**

Crea un pathnode que corresponde a un join nestloop entre dos relaciones.

| | |
|---|---|
| 'joinrel' | es la relación del join. |
| 'jointype' | es el tipo de join requerido |
| 'outer_path' | es la path externa |
| 'inner_path' | es la path interna |
| 'restrict_clauses' | son los nodos RestrictInfo a aplicarse en el join |
| 'pathkeys' | son las llaves de las path del nuevo path del join. |

Retorna el nodo path resultante.

- **create_mergejoin_path**

Crear uno pathnode que corresponde a una combinación mergejoin entre dos relaciones.

| | |
|---|---|
| 'joinrel' | es la relación join. |
| 'jointype' | es el tipo de join requerido. |
| 'outer_path' | es la path externa. |
| 'inner_path' | es la path interna. |
| 'restrict_clauses' | son los nodos RestrictInfo a aplicar en el join. |
| 'pathkeys' | son las llaves de paths del nuevo path del join. |
| 'mergeclauses' | son los nodos RestrictInfo a usar como cláusulas merge (este debe ser un subconjunto de la lista restrict_clauses). |
| 'outersortkeys' | son los sort varkeys de la relación externa |
| 'innersortkeys' | son los sort varkeys de la relación interna |

- **create_hashjoin_path**

Crea un pathnode que corresponde a un hash join entre dos relaciones.

| | |
|---|---|
| 'joinrel' | es la relación del join. |
| 'jointype' | es el tipo de join requerido. |
| 'outer_path' | es la path externa más barata. |
| 'inner_path' | es la path interna más barata. |
| 'restrict_clauses' | son los nodos de RestrictInfo a aplicarse en el join. |
| 'hashclauses' | son los nodos de RestrictInfo a utilizar como cláusulas hash (este debe ser un subconjunto de la lista restrict_clauses). |

**Archivo plancat.c**

Rutinas para acceder al catálogo del sistema.

Incluye las siguientes librerías:

postgres.h
math.h
access/genam.h
access/heapam.h
catalog/catname.h
catalog/pg_amop.h
catalog/pg_inherits.h
catalog/pg_index.h
nodes/makefuncs.h
optimizer/clauses.h
optimizer/plancat.h
optimizer/tlist.h
parser/parsetree.h
rewrite/rewriteManip.h
utils/builtins.h
utils/fmgroids.h
utils/lsyscache.h
utils/relcache.h
utils/syscache.h
catalog/catalog.h
miscadmin.h

Entre sus funciones se encuentran:

- **get_relation_info**

Recupera la información del catálogo para una relación dada.

Dado el Oid de la relación, retorna el Info siguiente en campos del struct de RelOptInfo:

| | |
|---|---|
| min_attr | el AttrNumber válido más bajo. |
| max_attr | el AttrNumber válido más alto. |
| indexlist | lista de IndexOptInfos para los índices de la relación. |
| pages | número de páginas. |
| tuples | número de tuplas. |

- **build_physical_tlist**

Construye un targetlist consistente de exactamente los atributos del usuario de la relación, en orden. El ejecutor puede - especial-caso - tales tlists evitar una proyección en paso de tiempo de pasada, así se utiliza tales tlists preferenciales para la exploración de nodos.

- **restriction_selectivity**

Retorna la selectividad de una cláusula específica del operador de restricción. Este código ejecuta los procedimientos registrados almacenados en la relación del operador, llamando la función encargada.

Vea clause_selectivity() para el significado de los parámetros adicionales.

- **join_selectivity**

Retorna la selectividad de la cláusula específica del operador join. Este código ejecuta los procedimientos registrados almacenados en la relación del operador, llamando la función encargada.

- **find_inheritance_children**

Retorna una lista que contiene el OIDs de todas las relaciones que herede "directamente" de la relación con OID ' inhparent'.

Nota: pudo ser una buena idea crear un campo inhparent pg_inherits en el índice, de modo que aquí se pueda utilizar un indexscan en vez de la exploración secuencial. Sin embargo, en bases de datos típicas los pg_inherits no tendrán bastantes entradas para justificar un indexscan.

- **has_subclass**

En la implementación actual, retornaría has_subclass si una clase particular podría tener una subclase. No retornará el resultado correcto si una clase tenía una subclase que fue caída más adelante. Esto es porque relhassubclass en pg_class no son actualizados cuando una subclase se cae, sobre todo debido a intereses de concurrencia.

Actualmente has_subclass se utilizan solamente como corte de la eficacia para saltar las búsquedas innecesarias de la herencia, así que ésta es ACEPTABLE.

- **has_unique_index**

Detecta si hay un índice único en el atributo especificado de la relación especificada, así permite que se concluya que todos los valores (no nulos) del atributo son distintos.

**Archivo restrictinfo.c**

RestrictInfo nodo de manipulación de rutinas.

Incluye las siguientes librerías:

postgres.h
optimizer/clauses.h
optimizer/paths.h
optimizer/restrictinfo.h
optimizer/var.h

Entre sus funciones se encuentran:

- **restriction_is_or_clause**

Retorna t si el nodo restrictinfo contiene una cláusula ' or '.

- **get_actual_clauses**

Retorna una lista que contiene las cláusulas vacías 'restrictinfo_list'.

- **get_actual_join_clauses**

Cláusulas de extracción del 'restrictinfo_list', separando los que emparejen sintácticamente el nivel del join de los que fueron empujadas hacia abajo.

- **remove_redundant_join_clauses**

Dado una lista de cláusulas RestrictInfo que deben ser aplicadas en un join, quite cualquier cláusula duplicada o redundante.

Se debe eliminar duplicados al formar el restrictlist para un joinrel, puesto que se verá muchas de las mismas cláusulas que llegan de ambos relaciones de entrada. También, si una cláusula es una cláusula mergejoinable, es posible que sea redundante con cláusulas anteriores (véase optimizer/README para discusión). Se detecta ese caso y se omite la cláusula redundante de la lista del resultado.

El resultado es una lista fresca, pero señala a los mismos nodos miembro que estaban en la entrada.

- **select_nonredundant_join_clauses**

Dado una lista de cláusulas RestrictInfo que deben ser aplicadas en un join, seleccione los que no son redundantes con alguna cláusula en reference_list.
Esto es similar a los remove_redundant_join_clauses, pero se busca las redundancias con una lista separada de cláusulas (es decir, cláusulas que tienen aplicado ya debajo del join mismo).

Observe que se asume que el restrictinfo_list dado ya se ha comprobado para saber si hay redundancias locales, así que no se comprueba otra vez.

- **join_clause_is_redundant**

Retorna verdadero si el rinfo es redundante con alguna cláusula en reference_list.

Ésta es la rutina principal de remove_redundant_join_clauses y los select_nonredundant_join_clauses.

Se puede detectar las cláusulas mergejoinable redundantes muy barato usando los pathkeys izquierdos y derechos, de los cuales identifique únicamente los grupos de variables equijoined en cuestión. Todos los miembros de un grupo pathkey que están en la relación izquierda se han forzado ya para ser iguales; asi mismo aquellos en la relación derecha. Así pues, se necesita tener solamente una cláusula que compruebe igualdades entre algún miembro del grupo a la izquierda y cualquier miembro a la derecha; por transitividad, todo el resto es entonces igual.

Sin embargo, cláusulas que están de la forma "var expr = const expr" no pueden ser eliminadas como redundantes. Esto es porque cuando hay expresiones constantes en un grupo pathkey, generate_implied_equalities() suprime cláusulas "var = var" en favor de cláusulas "var = const". No se puede permitir dejar alguno a lo último, aunque puede ser que se parezcan redundantes por los test del miembro del pathkey.

Caso especial extraño: si se tiene dos cláusulas que parezcan redundantes a menos que una se empuje hacia abajo en el join externo y el otro no, entonces no son realmente redundantes, porque una obliga a las filas combinadas adicionarse después de las filas nulas ocupadas, y no lo hace el otro.

**ANALISIS EXECUTOR**

La consulta al llegar al Executor es tomada por las fases: *inicialización, ejecución y finalización*.

- *Fase de Inicialización*: En esta fase se crean e inicializan las estructuras que se asocian a los nodos del plan de ejecución, se abren los archivos correspondientes a las relaciones que intervienen a la consulta y se asigna memoria a cada proceso e inicialización de variables. Esta etapa se realiza mediante la función *ExecutorStart* del archivo *ExecMain.c*, desde su interior se llama a la función *InintPlan* que a su vez invoca *ExecInitNode* del archivo *ExecProcNode.c*.
- *Fase de Ejecución*: Esta fase realiza la ejecución real de la consulta. Esta etapa se realiza mediante la función *ExecRun* del archivo *ExecMain.c*.
- *Fase de Finalización*: Esta fase se encarga de liberar todos los recursos que se reservaron en la fase de inicialización, esto lo realiza la función *ExecutorEnd* en el archivo *ExecMain*.

Teniendo en cuenta la fase de inicialización, ejecución y fase de finalización se analizarán los archivos mencionados anteriormente con sus respectivas funciones.

**Archivo Execmain.c**

Interfaz (rutinas) de Ejecutor de nivel superior.

RUTINAS DE LA INTERFAZ
- ExecutorStart()
- ExecutorRun()
- ExecutorEnd()

El viejo ExecutorMain () ha sido reemplazado por ExecutorStart (), ExecutorRun () y ExecutorEnd (). Estos tres procedimientos son las interfaces externas para el ejecutor. En cada caso, el descriptor del query y el estado de ejecución son requeridos como argumentos

ExecutorStart() es llamado al empezar cualquier ejecución de algún query plan y ExecutorEnd() debería siempre ser llamado al final de la ejecución de un plan. ExecutorRun acepta dirección y count como argumentos que especifican cuando el plan esta siendo ejecutado forwards, backwards, y para cuántas tuplas.

Incluye dentro de su código las siguientes librerías (entre otras):

postgres.h
access/heapam.h
catalog/heap.h
catalog/namespace.h
commands/tablecmds.h
commands/trigger.h
executor/execdebug.h
executor/execdefs.h
miscadmin.h
optimizer/var.h
parser/parsetree.h
utils/acl.h
utils/lsyscache.h

Entre sus funciones se encuentran:

- **ExecutorStart**

Esta rutina debe ser llamada al empezar cualquier ejecución de algún query plan.
Retorna un TupleDesc que describe los atributos de los tuplas para ser retornado por el query (El mismo valor guardado en queryDesc).

Nota: El CurrentMemoryContext cuando es llamado debe ser el contexto para ser usado como el contexto per-query por el query plan. ExecutorRun() y ExecutorEnd () deben ser llamados en este mismo contexto de memoria.

- **ExecutorRun**

Esta es la rutina principal del módulo del ejecutor. Acepta el descriptor del query del trafic cop (policía de tráfico) y ejecuta el query plan. ExecutorStart ya debe haber sido llamado.

Si la dirección es NoMovementScanDirection entonces no se hace nada, como no sea para poner en marcha cierra el destino. De otra manera recupera para 'contar' tuplas en la dirección especificada.

- **ExecutorEnd**

Esta rutina debe ser llamada al finalizar la ejecución de cualquier query plan. Esta función junto con las 2 anteriores son las más importantes de este módulo.

- **ExecCheckQueryPerms**

Revisa permisos de acceso para todas las relaciones referenciadas en el query.

- **ExecCheckPlanPerms**

Hace un escaneo recursivo del plan tree para revisar permisos de acceso en los subplanes.

- **ExecCheckRTPerms**

Revisa los permisos de acceso para todas las relaciones listadas en un range table.

- **ExecCheckRTEPerms**

Revisa permisos de acceso para una sola relación.

- **InitPlan**

Inicializa el query plan: Abre archivos, ubica almacenamientos y comienza el manejador de reglas.

- **initResultRelInfo**

Inicializa ResultRelInfo en caso de una relación resultado.

- **EndPlan**

Limpia el query plan. Cierra archivos y libera almacenamientos.

- **ExecutePlan**

Procesa el query plan para recuperar 'numberTuples' tuplas en la dirección especificada. Recupera todos las tuplas si numberTuples es 0. result es un slot conteniendo la última tupla en el caso de un SELECT o NULL de otra manera.
Nota: El atributo ctid es un 'junk' que es removido antes de que el usuario lo pueda ver.

- **ExecSelect**

SELECTs son fáciles. Solo pasa la tupla a la función print apropiada. La única complejidad es cuando se presenta un “SELECT INTO", en cuyo caso se inserta la tupla en la relación apropiada (nota: Esta es una relación recién creada así es que es necesario preocuparse por índices o bloqueos).

- **ExecInsert**

INSERTs son mas 'tramposos'. Se tiene que insertar la tupla en la relación base e insertar las tuplas en los índices de las relaciones.

- **ExecDelete**

DELETE es similar al UPDATE. Borra las tuplas y sus índices.

- **ExecUpdate**

Nota: no se puede correr UPDATE con transacciones off porque los UPDATEs son de hecho INSERTs y el escaneo quedará en un loop infinito. Actualizando una tupla al insertar debería arreglarlo pero es probable quedarse atorados en un loop infinito que dañe la base de datos.

- **EvalPlanQual**

Revisa una tupla modificada para mirar si se quiere procesar su versión actualizada bajo las reglas READ COMMITTED rules. Revise backend/executor/README para mas información de como trabaja esta función.

**Archivo ExecProcnode.c**

Contiene funciones de despacho que llaman las rutinas apropiados "initialize", "get a tuple", y "cleanup" para el tipo dado del nodo. Si el nodo tiene a los hijos, entonces luego probablemente llamará a ExecInitNode, ExecProcNode, o ExecEndNode en sus subnodos y hacen el procesamiento apropiado

RUTINAS DE LA INTERFAZ

- ExecInitNode: inicializa un nodo plan y sus subplanes.
- ExecProcNode: Obtiene una tupla ejecutando el nodo del plan.
- ExecEndNode: Cierra un nodo del plan y sus subplanes.
- ExecCountSlotsNode. Cuenta los slot tupla necesitadas por el plan tree.
- ExecGetTupType: Obtiene el tipo de tupla del resultado de un plan nodo.

NOTAS
En versiones anteriores, fueron tres archivos. Ahora esta todo combinado en un archivo a fin de que sea más fácil conservar ExecInitNode, ExecProcNode, y ExecEndNode sincronizados cuando los nuevos nodos se agregan.

Incluye dentro de su código las siguientes librerías (entre otras):

postgres.h
executor/executor.h
executor/instrument.h
executor/nodeAgg.h
executor/nodeAppend.h
executor/nodeGroup.h
executor/nodeHash.h
executor/nodeHashjoin.h
executor/nodeIndexscan.h
executor/nodeTidscan.h
executor/nodeLimit.h
executor/nodeMaterial.h
executor/nodeMergejoin.h
executor/nodeNestloop.h
executor/nodeResult.h
executor/nodeSeqscan.h
executor/nodeSetOp.h
executor/nodeSort.h
executor/nodeSubplan.h
executor/nodeSubqueryscan.h
executor/nodeFunctionscan.h
executor/nodeUnique.h
miscadmin.h

tcop/tcopprot.h

Entre sus funciones se encuentran:

- **ExecInitNode**

Recursivamente inicializa todos los nodos en el plan.
Initial States: 'node' es el plan producido por el query planner. Retorna TRUE/FALSE si el plan fue exitosamente inicializado.

- **ExecProcNode**

Estado inicial: el query tree debe ser inicializado una vez por la llamada a ExecInit.

- **ExecEndNode**

Recursivamente limpia todos los nodos en el plan. Después de esta operación, el query plan no podrá procesar nada en adelante. Esta función debería ser llamada solo después de que el query plan ha sido totalmente ejecutado.

- **ExecGetTupType**

Esto función da una descripción de la tupla, para las tuplas retornadas por este nodo. Realmente se desea que pueda descartar esta rutina, pero desde que no todos los nodos almacenan su tipo de información en el mismo lugar, se tiene que hacer algo especial para cada tipo de nodo.

**Archivo nodeSeqscan.c**

Soporta rutinas para el escaneo secuencial de relaciones.

RUTINAS DE LA INTERFAZ
ExecSeqScan: Escaneo secuencial de una relación.
ExecSeqNext: Retorna la siguiente tupla en orden secuencial.
ExecInitSeqScan: Crea e inicializa un nodo seqscan.
ExecEndSeqScan: Libera cualquier almacenamiento ubicado a través de las rutinas de C.
ExecSeqReScan: Escanea nuevamente la relación.
ExecMarkPos: Marca la posición de escaneo.
ExecRestrPos: Restaura la posición de escaneo.

Incluye dentro de su código las siguientes librerías (entre otras):

postgres.h
access/heapam.h
executor/execdebug.h
executor/nodeSeqscan.h
parser/parsetree.h

Entre sus funciones se encuentran:

- **SeqNext**

Función muy utilizada por ExecSeqScan.

- **ExecSeqScan**

Escanea la relación secuencialmente y retorna la siguiente tupla. Llama la rutina ExecScan( ) y le pasa a ella el método de acceso el cuál recupera tuplas secuencialmente.

- **InitScanRelation**

Este hace la inicialización para escanear relaciones y los subplanes.

## ANEXO B

## PRINCIPALES BIBLIOTECAS DE POSTGRES

**Generales**

- postgres.h

Archivo primario incluido para los archivos .c del servidor PostgreSQL. Este seria el primer archivo incluido por los módulos del backend de PostgreSQL pero el código del cliente incluiría postgres_fe.h no postgres.h.

- math.h

Librería estándar de C para operaciones y cálculos matemáticos.

- miscadmin.h

Contiene administración de postgres en general e inicialización.

**Access**

- access/genam.h

Definiciones del método de acceso del índice generalizado de POSTGRES.

- access/heapam.h

Definición del método de acceso de pila de POSTGRES.

**Catalog**

- catalog/catalog.h

Prototipos para funciones en backend/catalog/catalog.c.

- catalog/catname.h

Definiciones del nombre de la relación del catálogo del sistema de POSTGRES.

- catalog/heap.h

Prototipos para funciones en backend/catalog/heap.c.

- catalog/namespace.h

Prototipos para funciones en backend/catalog/namespace.c.

- catalog/pg_amop.h

Definición de la relación "amop" del sistema junto con el contenido inicial de las relaciones

- catalog/pg_inherits.h

Definición de la relación "heredada" del sistema junto con el contenido inicial de las relaciones.

- catalog/pg_index.h

Definición de la relación "índice" del sistema junto con el contenido inicial de las relaciones.

- catalog/pg_operator.h

Definición del operador de relaciones de sistema (pg_operator) junto con los contenidos iniciales de las relaciones.

- catalog/pg_statistic.h

Definición de la relación "estadística" del sistema junto con el contenido inicial de las relaciones.

- catalog/pg_type.h

Definición del "tipo" de relación de sistema (pg_type) junto con los contenidos iniciales de la relación.

**Commands**

- commands/tablecmds.h

Prototipos para tablecmds.c.

- commands/ trigger.h

Declaraciones para el manejo de trigger.

**Executor**

- executor/executor.h

Soporte para el módulo ejecutor de POSTGRES.

- executor/ execdebug.h

#defines que gobiernan el comportamiento del depurador en el ejecutor.

- executor/ instrument.h

Definiciones para la colección de estadísticas de tiempos de ejecución.

- executor/nodeGroup.h

Prototipos para nodeGroup.c.

- executor/nodeHash.h

Prototipos para node Hash.c.

**Nodes**

- nodes/makefuncs.h

Presenta prototipos para el creador de funciones (para nodos primitivos).

- nodes/nodeFuncs.h

- nodes/plannodes.h

Definición de nodos para el query plan.

**Optimizer**

- optimizer/cost.h

Prototipos para costsize.c y clausesel.c.

- optimizer/clauses.h

Presenta prototipos para el clauses.c.

- optimizer/joininfo.h

Librería de prototipos para joininfo.c.

- optimizer/pathnode.h

Librería de prototipos para pathnode.c y relnode.c.

- optimizer/paths.h

Prototipos para varios archivos en el optimixzer/path (estos eran cabeceras de archivos separados).

- optimizer/plancat.h

Prototipos para plancat.c.

- optimizer/planmain.h

Presenta prototipos para varios archivos en el optimizer/plan.

- optimizer/planner.h

Prototipos para planner.c.

- optimizer/prep.h

Prototipos para los archivos en optimizer/prep.

- optimizer/geqo.h

Prototipos para varios archivos en optimizer/geqo.

- optimizer/restrictinfo.h

Prototipos para restrictinfo.c.

- optimizer/tlist.h

Prototipos para tlist.c.

- optimizer/var.h

Prototipos para var.c.

**Parser**

- parser/analyze.h

Se definen los objetos necesarios para realizar el correspondiente análisis del parse tree y su esquema.

- parser/gramparse.h

Presenta Declaraciones para rutinas exportadas desde lexer y archivos del parser.

- parser/parsetree.h

Presenta Rutinas para acceder varios componentes y subcomponentes de los parse trees, como conseguir un RTE y un número de atributo, retornar el nombre apropiado de la variable o alias para ese atributo que pertenece a ese RTE. Este también es el encargado de retornar el Id de la relación en el proceso de transformación.

- parser/parse_agg.h y parser/parse_clause.h

Estas librerías Definen funciones, parámetros, flags, listas, índices, entre otros, para el tratamiento de los nodos del parse tree.

- parser/parse_coerce.h

Define rutinas para el tipo de coerción de RTE´s y sus atributos. Por ejemplo: si es booleana, de tipo cadena, numérica, definida por el usuario, etc.

- parser/parse_func.h

Esta estructura es usada para explorar la sucesión jerárquica sobre los nodos en el tipo de árbol en orden de desambiguar funciones polimórficas.

- parser/parse_oper.h

Presenta rutinas para mirar los operadores, las entradas y sus tipos exactos, presentes en una operación que requieran o no requieran coerción.

- parser/parse_relation.h

Presenta los prototipos para el soporte de rutinas que interactúan con relaciones dentro del parser.

- parser/parse_target.h

Maneja las target lists.

**Rewrite**

- rewrite/rewriteManip.h

Subrutinas de manipulación del Querytree para reescritura de consultas.

**Tcop**

- tcop/tcopprot.h

Prototipos para postgres.c.

**Utils**

- utils/acl.h

Definición de (y soporte para) control de acceso de estructuras de datos de lista.

- utils/builtins.h

Declaraciones para tipos de estructuras en operaciones

- utils/fmgroids.h

Esta librería es mencionada pero no existe en esta versión de Postgres.

- utils/datum.h

POSTGRES rutinas de manipulación de Datum (Tipos de datos abstractos). Estas rutinas son manejadas por la información de 'typbyval' y 'typlen', los que deben previamente haber sido obtenidos por el llamador del datatype del Datum. (Se hace esto porque en la mayoría de las situaciones el llamador puede bloquear la información solo una vez y usarla para muchas operaciones por-datum)

- utils/lsyscache.h

Rutinas convenientes para queries comunes en la cache de catálogo de sistema.

- utils/memutils.h

Contiene declaraciones para funciones utilitarias para el manejo de memoria.

- utils/relcache.h

Definiciones de descriptor cache de la relación.

- utils/selfuncs.h

Funciones de selectividad y funciones de estimación de costo del índice para operadores estándar y métodos de acceso del índice.

- utils/syscache.h

Definiciones de la cache del catálogo de sistema.

# ANEXO C

# LEX Y YACC

## Que Son y Para Que Sirven

Lex y Yacc son un par de especificaciones que sirven para generar tokenizers y parsers en C que reconozcan gramáticas libres de contexto, como lenguajes de programación o calculadoras entre otros.

Lex es el encargado de leer de la entrada, típicamente stdin y extraer de la misma los tokens reconocidos por el basado en un lenguaje de expresiones regulares.

Yacc sirve para generar parsers, usa a lex para leer y reconocer sus tokens y basado en reglas sencillas en formato similar al BNF, es capaz de ir reduciendo una expresión con bastante eficiencia.

## Principales Implementaciones

Ambos, Lex y Yacc, fueron desarrollados en los 70's en los laboratorios Bell de AT&T, y estuvieron disponibles desde la 7a Edición de UNIX, versiones antiguas derivadas de BSD seguían usando Lex y Yacc de AT&T, hasta la aparición de flex y bison (Análogos a lex y yacc respectivamente) que cuentan con algunas características extra además de las tradicionales, así como un mejor soporte para reducciones o expresiones muy largas o complejas.

## LEX

### Conceptos Básicos

Se puede definir a lex como una herramienta para construir analizadores léxicos o "lexers". Un lexer lee de un flujo de entrada cualquiera, y la divide en unidades léxicas (la tokeniza), para ser procesada por otro programa o como producto final.

Para escribir una especificación léxica en lex, es necesario crear un conjunto de patrones (Expresiones Regulares), mismos, que cuando el programa este completo, van a ser reconocidos como tokens o unidades léxicas.

Lex no produce un programa compilado, lo que hace, es traducir esa especificación a C, incluyendo una rutina llamada yylex(), que es la usada para iniciar en análisis de la entrada.

La entrada es tomada de yyin, que por defecto su valor es stdin, es decir, la pantalla o terminal, pero este valor puede ser modificado por cualquier apuntador a un archivo.

También es posible leer la entrada desde un arreglo de caracteres u otros medios, para cual es necesario implementar algunas funciones de lex mismas que se definirán en la última parte de esta sección (Agregar Funcionalidad).

**Expresiones Regulares**

Para poder crear expresiones regulares y patrones para las reglas, es necesario saber que la concatenación de expresiones se logra simplemente juntando dos expresiones, sin dejar espacio entre ellas y que es bueno declarar una expresión muy compleja por partes como definiciones, y así evitar tener errores difíciles de encontrar y corregir.

A continuación una lista de las expresiones regulares mas usadas en lex.

| Ops | Ejemplo | Explicación |
|---|---|---|
| [ ] | [a-z] | Una clase de Caracteres, coincide con un carácter perteneciente a la clase, pueden usarse rangos, como en el ejemplo, cualquier carácter, excepto aquellos especiales o de control son tomados literalmente, en el caso de los que no, pueden usarse secuencias de escape como las de C, \t, \n etcétera. |
| * | [ \n\t]* | Todas las cadenas que se puedan formar, se puede decir que este operador indica que se va a coincidir con cadenas formadas por ninguna o varias apariciones del patrón que lo antecede.<br>El ejemplo coincide con cualquier combinación de símbolos usados para separar, el espacio, retorno y tabulador. |
| + | [0-9]+ | Todas las cadenas que se puedan formar, excepto cadenas vacías.<br>En el ejemplo se aceptan a todos los números naturales y al cero. |
| . | .+ | Este es una expresión regular que coincide con cualquier entrada excepto el retorno de carro ("\n").<br>El ejemplo acepta cualquier cadena no vacía. |
| { } | a{3,6} | Indica un rango de repetición cuando contiene dos números separados por comas, como en el ejemplo, la cadena aceptada será aquella con longitud 3, 4, 5 o 6 formada por el carácter 'a'. |
| ? | -?[0-9]+ | Indica que el patrón que lo antecede es opcional, es decir, puede existir o no.<br>En el ejemplo, el patrón coincide con todos los números enteros, positivos o negativos por igual, ya que el signo es opcional. |
| \| | (-\|+\|~)?[0-9]+ | Este hace coincidir, al patrón que lo precede o lo antecede y puede usarse consecutivamente.<br>En el ejemplo tenemos un patrón que coincidirá con un entero positivo, negativo o con signo de complemento. |
| "" | "bye" | Las cadenas encerradas entre " y " son aceptadas literalmente, es decir tal como aparecen dentro de las comillas, para incluir caracteres de control o no imprimibles, pueden usarse dentro de ellas secuencias de escape de C.<br>En el ejemplo la única cadena que coincide es 'bye'. |

| | | |
|---|---|---|
| \ | \. | Indica a lex que el carácter a continuación será tomado literalmente, como una secuencia de escape, este funciona para todos los caracteres reservados para lex y para C por igual.<br>En el ejemplo, el patrón coincide solo con el carácter "." (punto), en lugar de coincidir con cualquier carácter, como seria el casi sin el uso de "\". |
| <<EOF>> | [a-z] | Solo en flex, este patrón coincide con el fin de archivo. |

**Agregar Funcionalidad**

Es posible hacer que el lexer se comporte un tanto diferente de los defaults en cuanto a la implementación se refiere, redefiniendo las funciones que el lexer usa, algunas de las cosas que se pueden hacer es cambiar la entrada, modificar el manejo de final de archivo, etc.

Pero antes de poder hacer esto, es necesario repasar algunas variables y funciones, que se usan dentro de un programa generado por lex.

| Prototipo | Descripción |
|---|---|
| char *yytext; | Contiene el token que acaba de ser reconocido, su uso es principalmente dentro de las reglas, donde es común hacer modificaciones al token que acaba de ser leído o usarlo con algún otro fin. |
| int yyleng; | Contiene la longitud del token leído, su valor es equivalente a yyleng = strlen(yytext);. |
| FILE *yyin; | Es un apuntador del que se leen los datos, si este no se modifica, su valor por defecto es stdin. |
| FILE *yyout; | Es un apuntador a la salida por default del programa, su valor predefinido es stdout. |
| int input(void); | Esta es en realidad una macro, cuya función es alimentar al tokenizer cada vez que se le llama, esta regresa el siguiente carácter de la entrada. |
| void unput(int); | Esta macro hace lo contrario a input(), esta pone el carácter especificado como argumento de regreso a la entrada del flujo. |
| void output(int); | Esta macro, escribe su argumento en yyout. |
| int yyinput(void); | Es una interfaz para la macro input(). |
| void yyunput(int); | Es una interfaz para la macro unput(). |
| void yyoutput(int); | Es una interfaz para la macro output(). |
| int yywrap(void); | Esta función sirve para manejar las condiciones de fin de archivo, cada vez que el lexer llega a un fin de archivo, llama a esta función para saber que hacer, si regresa 0, entonces sigue leyendo de la entrada, si es 1 el lexer regresa un token nulo para indicar que se llego al fin de archivo. |
| int yylex(void); | Esta es la función principal de un lexer, para añadir código a esta función, es necesario, incluirlo en la sección de reglas encerrado entre %{ y %}. |

**Reimplementaciones y Usos más Comunes**

***FILE *yyin***
Este es un apuntador declarado globalmente que apunta al lugar de donde se van a leer los datos, por ser un file pointer, este solo puede leer de flujos como archivos, para leer de una cadena es necesario reimplementar el macro input().

***FILE *yyout***
Este es el lugar al que se escriben por default todos los mensajes, al igual que yyin esta declarado globalmente y es un apuntador.

***int input(void)***
El objetivo de esta Macro es alimentar a yylex() carácter por carácter, devuelve el siguiente carácter de la entrada, la intención más común para modificar esta función, es cambiar el origen de la entrada de manera mas flexible que con yyin, ya que no solo es posible leer de otro archivo, sino que también es posible leer el flujo para parsear una cadena cualquiera, o un grupo de cadenas como una línea de comandos.

***void unput(int)***
El objetivo de esta macro, es regresar un carácter a la entrada de datos, es útil para yylex() tener una de estas, ya que para identificar un patrón puede ser necesario saber que carácter es el que sigue. La intención de reimplementar esta es complementar el uso de la reimplementacion de input(), ya que input() y unput() deben ser congruentes entre si.

***int yywrap(void)***
Esta función, es auxiliar en el manejo de condiciones de final de archivo, su misión es proporcionarle al programador la posibilidad de hacer algo con estas condiciones, como continuar leyendo pero desde otro archivo.

***int yylex(void)***
Esta función, es casi totalmente implementada por el usuario en la sección de reglas, donde puede agregarse código encerrado entre %{ y %} así como en las reglas mismas.

## YACC

### Conceptos Básicos

Lex permite generar un analizador léxico, un lexer o tokenizer, que puede reconocer patrones de expresiones regulares y en un momento dado determinar que es lo que se esta leyendo, pero para aplicaciones un poco más complejas, también puede ser necesario, analizar gramaticalmente la composición de la entrada para en un momento dado, determinar si la entrada coincide o no con una gramática definida y resolverla, o darle algún significado un tanto mas complejo.

Es correcto decir que yacc es una herramienta que sirve para generar un programa, capaz de analizar gramaticalmente una entrada dada por lex, a partir de una especificación. Esta especificación, debe contener los tokens reconocidos y los tipos de datos de los mismos si es que se ocupan para realizar operaciones sobre ellos, y una especificación de gramática en un formato similar a BNF (Backus Naus Form), que va desde el símbolo no terminal más general a cada una de las opciones terminales.

Una especificación yacc se divide en tres secciones diferentes de manera similar a lex, la de definiciones, la de reglas, y la de subrutinas, que van igualmente separadas por un '%%', mismas que pueden incluir código de C encerrado entre un %{ y un %}.

### Definiciones

En esta primera sección, al igual que en lex, incluimos las librerías que usaremos en el programa, definiciones de los tokens, tipos de datos y precedencia de la gramática.

#### *%union*

Esta definición, se traduce a una unión de C que a su vez dará el tipo de dato a una variable global de nombre yylval que será de donde yacc tomara los datos a procesar, en la unión se definen miembros cuyos correspondientes tipos de datos serán usados para dar el tipo de dato a los tokens

#### *%token y %type*

%token sirve para definir los tokens que hay, y si es necesario, el tipo de dato que usan, todos los tokens son tomados como símbolos terminales, lo cual veremos mejor reflejado en la sección de reglas, estos también tienen el objetivo de servir como etiquetas que yylex() regresa a yacc para identificar el token que se ha leído recientemente.
Su uso es como sigue: %type es análogo a %token, solo que este define el tipo de dato para símbolos no terminales de la gramática, la única diferencia es que el tipo de dato a usar es obligatorio.

#### *%left y %right*

Permiten definir el tipo de precedencia de los tokens operadores, en este punto tenemos dos factores, la precedencia por si misma, y la agrupación de los operadores.
%left y %right indican si el operador se agrupa a la derecha o a la izquierda.

***%start***
En algunos casos es conveniente indicarle a yacc cual es el símbolo (no terminal) inicial a la hora de hacer el parser, es decir, el símbolo que se trata de reducir, si esta opción no es especificada, yacc toma al primer símbolo de la sección de reglas como símbolo inicial.

**Reglas**
En esta parte está la definición de la gramática, acá se observa que cada símbolo se define con su nombre, seguido de dos puntos ":" seguidos de varios símbolos que conformaran su composición gramatical que en caso de tener varias opciones, son separados por "|" (or) indicando que se tienen varias opciones para reducir ese símbolo y para terminar cada regla, un ";".

***Reducción***
Yacc reduce sus reglas generando un parser tree (no literalmente), y va resolviendo cada regla completa tan pronto como puede, lo cual trae un detalle de diseño de gramáticas en yacc, y es la diferencia entre especificar la recursividad por la derecha o por la izquierda, para expresiones muy sencillas que generen un parser tree pequeño no hay ningún problema pero para casos donde la reducción es compleja, puede desbordar la pila ya que cuando la recursión es derecha, para resolverla, tiene que guardar los datos de la izquierda, y si estos son demasiados, no puede manejarlos.

Por lo contrario, cuando la recursión es izquierda, no tiene que guardar datos que no va a utilizar por que recorre el árbol de izquierda a derecha y resuelve las reglas tan pronto como puede.

**Subrutinas**
En esta última sección, es posible reimplementar, siguiendo la misma idea de lex, algunas funciones que pueden ser útiles en algún momento dado o declarar nuevas funciones para usar.

Las funciones mas comúnmente implementadas son main() e yyerror(), la primera se usa para personalizar el programa con mensajes antes o después del parser, o para llamarlo varias veces en el código y la segunda la ocupa yyparse() cada vez que encuentra un error de sintaxis.

## ANEXO D

## EJEMPLOS DE ASOCIACIÓN Y CLASIFICACIÓN

### D1. Ejemplo de Asociación.

Sea la tabla estudiantes (programa, edad, sexo, estrato, promedio) de la figura D1.1 la cual esta discretizada, descubrir las Reglas de Asociación unidimensionales con un soporte mínimo de 2, un tamaño de regla de 2 y con una confianza mínima de 50%, para los itemsets de tamaño 1 a 3 formados por lo atributos programa, sexo, estrato.

**Figura D1.1. Tabla estudiantes**

| programa | edad | sexo | estrato | promedio |
|---|---|---|---|---|
| sistemas | 16-20 | m | 2 | medio alto |
| sistemas | 16-20 | f | 3 | regular |
| idiomas | 21-25 | f | 3 | regular |
| fisica | 21-25 | m | 2 | bajo |
| psicologia | 21-25 | f | 2 | alto |

La sentencia SQL que obtiene los itemsets frecuentes con el tamaño y soporte indicado es:

```
SELECT programa, sexo, estrato, count(*) AS soporte INTO assostudent
FROM estudiantes
ASSOCIATOR RANGE 1 UNTIL 3
GROUP BY programa, sexo, estrato HAVING count(*) >= 2
```

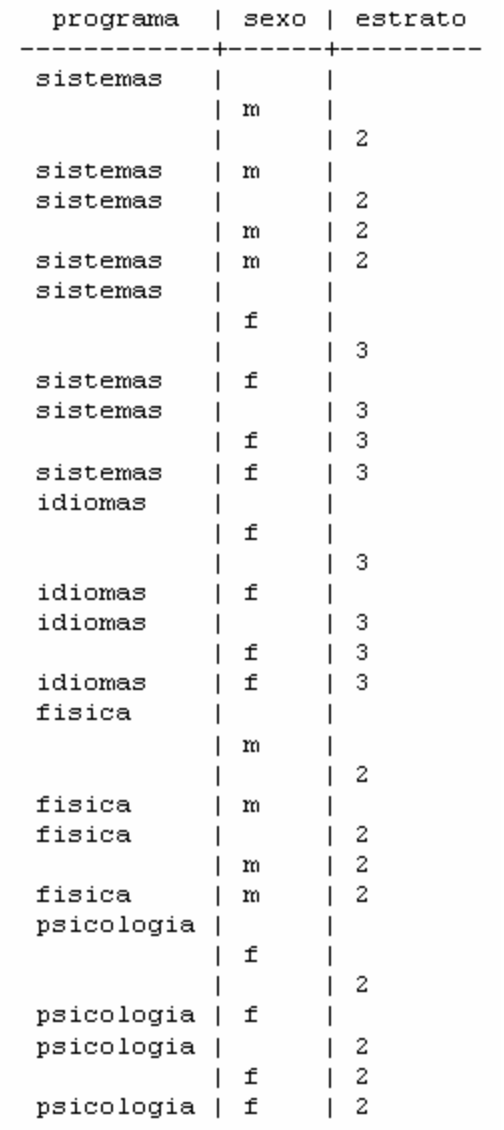
La sentencia SQL que genera las reglas de asociación unidimensionales con un tamaño de regla y confianza indicadas es:

```
SELECT * FROM DESCRIBE_ASSOCIATION_RULES ('assostudent',2,50)
```

El seguimiento para las sentencias SQL se realiza a continuación:

1. Inicialmente se ejecuta la cláusula SELECT. En este caso se seleccionan los atributos programa, sexo y estrato. Luego se ejecuta la cláusula ASSOCIATOR RANGE, es decir, se procede a generar los itemsets de tamaño 1 a 3 (Figura D1.2).

**Figura D1.2. Itemsets de tamaño 1 a 3 generados con la cláusula Associator Range sobre los atributos programa, sexo y estrato**



| programa | sexo | estrato |
|---|---|---|
| sistemas | | |
| | m | |
| | | 2 |
| sistemas | m | |
| sistemas | | 2 |
| | m | 2 |
| sistemas | m | 2 |
| sistemas | | |
| | f | |
| | | 3 |
| sistemas | f | |
| sistemas | | 3 |
| | f | 3 |
| sistemas | f | 3 |
| idiomas | | |
| | f | |
| | | 3 |
| idiomas | f | |
| idiomas | | 3 |
| | f | 3 |
| idiomas | f | 3 |
| fisica | | |
| | m | |
| | | 2 |
| fisica | m | |
| fisica | | 2 |
| | m | 2 |
| fisica | m | 2 |
| psicologia | | |
| | f | |
| | | 2 |
| psicologia | f | |
| psicologia | | 2 |
| | f | 2 |
| psicologia | f | 2 |

2. Se agrupa por programa, sexo y estrato de acuerdo a la cláusula GROUP BY y se calcula la función agregada count(). El resultado se almacena en la tabla assostudent (Figura D1.3).

**Figura D1.3. Tabla assostudent**

| programa | sexo | estrato | soporte |
|---|---|---|---|
| fisica | m | 2 | 1 |
| fisica | m | | 1 |
| fisica | | 2 | 1 |
| fisica | | | 1 |
| idiomas | f | 3 | 1 |
| idiomas | f | | 1 |
| idiomas | | 3 | 1 |
| idiomas | | | 1 |
| psicologia | f | 2 | 1 |
| psicologia | f | | 1 |
| psicologia | | 2 | 1 |
| psicologia | | | 1 |
| sistemas | f | 3 | 1 |
| sistemas | f | | 1 |
| sistemas | m | 2 | 1 |
| sistemas | m | | 1 |
| sistemas | | 2 | 1 |
| sistemas | | 3 | 1 |
| sistemas | | | 2 |
| | f | 2 | 1 |
| | f | 3 | 2 |
| | f | | 3 |
| | m | 2 | 2 |
| | m | | 2 |
| | | 2 | 3 |
| | | 3 | 2 |

3. Se obtiene los itemsets frecuentes de tamaño 1 a 3 que cumplan con un soporte mayor o igual a 2, el resultado sería como se muestra en la Figura D1.4.

**Figura D1.4. Tabla assostudent con soporte mayor o igual a 2**

| programa | sexo | estrato | soporte |
|---|---|---|---|
| sistemas | | | 2 |
| | f | 3 | 2 |
| | f | | 3 |
| | m | 2 | 2 |
| | m | | 2 |
| | | 2 | 3 |
| | | 3 | 2 |

4. Por último, el resultado de generar las reglas de tamaño 2 de la tabla assostudent con una confianza mínima del 50% se muestra en la figura D1.5.

**Figura D1.5. Reglas de asociación para assostudent**

| programa | sexo | estrato | n_regla | implica | soporte | confianza |
|---|---|---|---|---|---|---|
| - | f | - | 1 | A | 3 | 66.67 |
| - | - | 3 | 1 | C | 2 | |
| - | - | 3 | 2 | A | 2 | 100.00 |
| - | f | - | 2 | C | 3 | |
| - | m | - | 3 | A | 2 | 100.00 |
| - | - | 2 | 3 | C | 3 | |
| - | - | 2 | 4 | A | 3 | 66.67 |
| - | m | - | 4 | C | 2 | |

De la anterior tabla se pueden interpretar las siguientes reglas:

**Si** sexo(f) **entonces** estrato(3).
**Si** estrato(3) **entonces** sexo(f).
**Si** sexo(m) **entonces** estrato(2).
**Si** estrato(2) **entonces** sexo(m).

## D2. Ejemplo de Clasificación

Sea la tabla sintomas (d_cabeza, d_muscular, temperatura, gripa*)* de la figura D2.1, obtener el modelo de clasificación para los atributos condición d_cabeza, d_muscular, temperatura con el atributo clase gripa

**Figura D2.1. Tabla sintomas**

| d_cabeza | d_muscular | temperatura | gripa |
|---|---|---|---|
| no | si | alta | si |
| si | no | alta | si |
| si | si | media | no |
| no | si | normal | si |
| si | no | media | no |
| no | no | normal | no |
| si | no | normal | no |
| si | si | alta | si |

La sentencia SQL que forma todas las posibles combinaciones de los atributos condición con el atributo clase es:

```
SELECT d_cabeza, d_muscular, temperatura, gripa, count(*)
INTO clasesintomas
FROM sintomas
MATE BY d_cabeza, d_muscular, temperatura WITH gripa
GROUP BY d_cabeza, d_muscular, temperatura, gripa
```

La sentencia SQL que calcula la entropía para cada atributo condición es:

```
SELECT * FROM ENTRO('clasesintomas')
```

La sentencia SQL que calcula la ganancia de información para los atributos condición y cuyo resultado es la tabla que representa el árbol de decisión es:

```
SELECT * FROM GAIN('mate_entro')
```

El seguimiento para las sentencias SQL se realiza a continuación:

1. Se almacena en la tabla clasesintomas todas las ocurrencias de las diferentes combinaciones de los atributos d_cabeza, d_muscular, temperatura con el atributo gripa con la cláusula MATE BY (Figura D2.2).

**Figura D2.2. Tabla clasesintomas**

| d_cabeza | d_muscular | temperatura | gripa | count |
|---|---|---|---|---|
| no | no | normal | no | 1 |
| no | no | | no | 1 |
| no | si | alta | si | 1 |
| no | si | normal | si | 1 |
| no | si | | si | 2 |
| no | | alta | si | 1 |
| no | | normal | no | 1 |
| no | | normal | si | 1 |
| no | | | no | 1 |
| no | | | si | 2 |
| si | no | alta | si | 1 |
| si | no | media | no | 1 |
| si | no | normal | no | 1 |
| si | no | | no | 2 |
| si | no | | si | 1 |
| si | si | alta | si | 1 |
| si | si | media | no | 1 |
| si | si | | no | 1 |
| si | si | | si | 1 |
| si | | alta | si | 2 |
| si | | media | no | 2 |
| si | | normal | no | 1 |
| si | | | no | 3 |
| si | | | si | 2 |
| | no | alta | si | 1 |
| | no | media | no | 1 |
| | no | normal | no | 2 |
| | no | | no | 3 |
| | no | | si | 1 |
| | si | alta | si | 2 |
| | si | media | no | 1 |
| | si | normal | si | 1 |
| | si | | no | 1 |
| | si | | si | 3 |
| | | alta | si | 3 |
| | | media | no | 2 |
| | | normal | no | 2 |
| | | normal | si | 1 |

2. Se calcula la entropía como se hizo referencia en el capítulo 4 (Operadores algebraicos y Primitivas sql para el descubrimiento de conocimiento en Bases de Datos) para la tabla clasesintomas. La función ENTRO() almacena el resultado en la tabla mate*_entro* (Figura D2.4) y además crea la tabla *mate_values* (Figura D2.3) donde se almacenan los parámetros de transformación.

**Figura D2.3. Tabla mate_values**

| nro | atributo | valor | discret | nclases |
|---|---|---|---|---|
| 1 | d_cabeza | no | 0 | |
| 2 | d_cabeza | si | 1 | 2 |
| 3 | d_muscular | no | 0 | |
| 4 | d_muscular | si | 1 | 2 |
| 5 | temperatura | alta | 0 | |
| 6 | temperatura | media | 1 | |
| 7 | temperatura | normal | 2 | 3 |
| 8 | gripa | no | 0 | |
| 9 | gripa | si | 1 | 2 |

**Figura D2.4. Tabla mate_entro**

| d_cabeza | d_muscular | temperatura | gripa | count | entro |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 0 | 1 | 0 |
| 0 | 0 | -1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 2 | 1 | 1 | 0 |
| 0 | 1 | -1 | 1 | 2 | 0 |
| 0 | -1 | 0 | 1 | 1 | 0 |
| 0 | -1 | 2 | 0 | 1 | 500 |
| 0 | -1 | 2 | 1 | 1 | 500 |
| 0 | -1 | -1 | 0 | 1 | 528 |
| 0 | -1 | -1 | 1 | 2 | 390 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 2 | 0 | 1 | 0 |
| 1 | 0 | -1 | 0 | 2 | 390 |
| 1 | 0 | -1 | 1 | 1 | 528 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | -1 | 0 | 1 | 500 |
| 1 | 1 | -1 | 1 | 1 | 500 |
| 1 | -1 | 0 | 1 | 2 | 0 |
| 1 | -1 | 1 | 0 | 2 | 0 |
| 1 | -1 | 2 | 0 | 1 | 0 |
| 1 | -1 | -1 | 0 | 3 | 442 |
| 1 | -1 | -1 | 1 | 2 | 529 |
| -1 | 0 | 0 | 1 | 1 | 0 |
| -1 | 0 | 1 | 0 | 1 | 0 |
| -1 | 0 | 2 | 0 | 2 | 0 |
| -1 | 0 | -1 | 0 | 3 | 311 |
| -1 | 0 | -1 | 1 | 1 | 500 |
| -1 | 1 | 0 | 1 | 2 | 0 |
| -1 | 1 | 1 | 0 | 1 | 0 |
| -1 | 1 | 2 | 1 | 1 | 0 |
| -1 | 1 | -1 | 0 | 1 | 500 |
| -1 | 1 | -1 | 1 | 3 | 311 |
| -1 | -1 | 0 | 1 | 3 | 0 |
| -1 | -1 | 1 | 0 | 2 | 0 |
| -1 | -1 | 2 | 0 | 2 | 390 |
| -1 | -1 | 2 | 1 | 1 | 528 |

3. Para facilitar la construcción del árbol de decisión y por consiguiente la generación de reglas de clasificación se ejecuta la función GAIN() que almacena el árbol en la tabla *tclases* (Figura D2.5) y genera las reglas de decisión en la tabla *trulesclases* (Figura D2.6).

**Figura D2.5. Tabla tclases**

| NODO | PADRE | ATRIBUTO | VALOR | CLASE |
|---|---|---|---|---|
| 0 | -1 | 2 | -1 | -1 |
| 1 | 0 | 2 | 0 | 1 |
| 2 | 0 | 2 | 1 | 0 |
| 3 | 0 | 2 | 2 | -99 |
| 4 | 3 | 1 | 0 | 0 |
| 5 | 3 | 1 | 1 | 1 |

**Figura D2.6. Tabla trulesclases**

| id | atributo | valor |
|---|---|---|
| 1 | 2 | 0 |
| 1 | -777 | 1 |
| 2 | 2 | 1 |
| 2 | -777 | 0 |
| 3 | 1 | 0 |
| 3 | 2 | 2 |
| 3 | -777 | 0 |
| 4 | 1 | 1 |
| 4 | 2 | 2 |
| 4 | -777 | 1 |

Después de reemplazar los valores de transformación de la tabla trules (figura D2.5) por los valores originales utilizando la tabla mate_values (figura D2.3), se obtienen las tablas modificadas de las figuras D2.7 y D2.8. De la tabla tclases modificada (figura D2.7) se puede construir el árbol de decisión que se muestra en la figura D2.9.

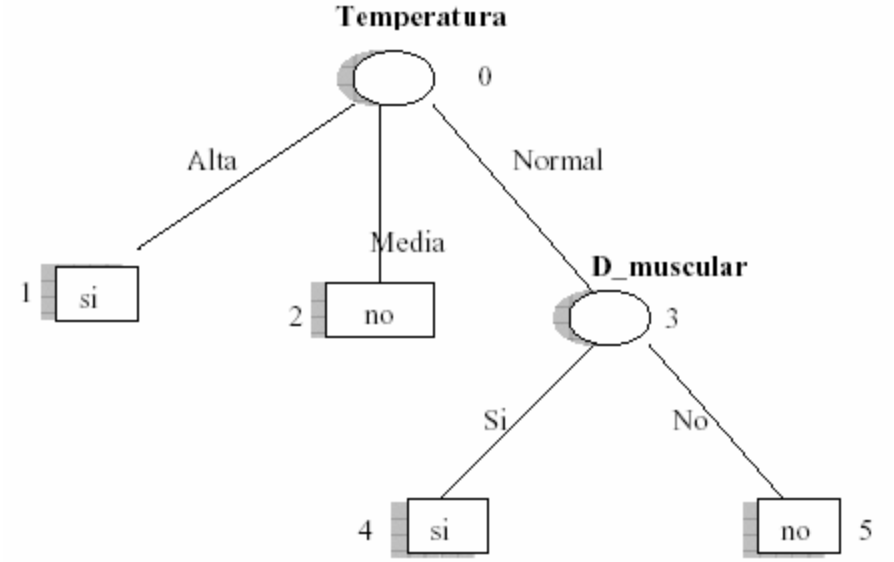
**Figura D2.7. Tabla tclases Modificada**

| NODO | PADRE | ATRIBUTO | VALOR | CLASE |
|---|---|---|---|---|
| 0 | NULL | temperatura | NULL | NULL |
| 1 | 0 | temperatura | alta | si |
| 2 | 0 | temperatura | media | no |
| 3 | 0 | temperatura | normal | -99 |
| 4 | 3 | d_muscular | no | no |
| 5 | 3 | d_muscular | si | si |

**Figura D2.8. Tabla trulesclases Modificada**

| id | atributo | valor |
|---|---|---|
| 1 | temperatura | alta |
| 1 | -777 | si |
| 2 | temperatura | media |
| 2 | -777 | no |
| 3 | d_muscular | no |
| 3 | temperatura | normal |
| 3 | -777 | no |
| 4 | d_muscular | si |
| 4 | temperatura | normal |
| 4 | -777 | si |

**Figura D2.9. Representación gráfica de tclases**




Las figuras D2.8 y D2.9 representan las siguientes reglas de clasificación:

**Si** Temperatura(alta) **entonces** gripa(si).
**Si** Temperatura(media) **entonces** gripa(no).
**Si** Temperatura(normal) y D_Muscular(no) **entonces** gripa(no).
**Si** Temperatura(normal) y D_Muscular(si) **entonces** gripa(si).

## ANEXO E

## INSTRUCCIONES DE INSTALACION DE POSTGRESQL-KDD

Este anexo describe la instalación del código fuente de PostgreSQL con las nuevas funciones de minería de datos:

### 1. Proceso de instalación de PostgreSQL-KDD

- Se ingresa como usuario root y se desplaza al directorio </usr/local/src/> donde se ubicaran las fuentes de postgresql-KDD.
  $ su
  # cd /usr/local/src/
  # cp /cdrom/postgresql-KDD.tar.gz.
- Se desempaqueta el archivo y se ingresa en el directorio generado.
  # tar xvfz postgresql-KDD.tar.gz
  # cd postgresql-7.3.4
- Antes de iniciar la instalación se configura el código fuente de la siguiente forma:
  # ./configure
- Para compilar e instalar, se digitan las siguientes secuencias de comandos:
  # gmake
  # gmake install
- Se crea el súper-usuario postgres.
  # adduser postgres
- Se crea el directorio de datos y de generación de archivos de seguimiento, y se le asigna permisos de propietario al usuario postgres.
  # mkdir /usr/local/pgsql/data
  # chown postgres:postgres /usr/local/pgsql/data
- Se inician las bases de datos básicas para el correcto funcionamiento de PostgreSQL y se ejecuta el servidor de PostgreSQL. Para esto, es necesario identificarse como usuario postgres.
  # su – postgres
  $ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
  $ /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data

### 2. Compilar las funciones de Asociación y Clasificación.

Para Asociación:

- Se ingresa como usuario root y se desplaza al directorio de asociación.

# cd /usr/local/src/postgresql-7.3.4/contrib/kdd/asociación

- Para compilar e instalar, se digitan las siguientes secuencias de comandos:
  # gmake
  # gmake install

Para Clasificación:

- Se ingresa como usuario root y se desplaza al directorio de clasificación.
  # cd /usr/local/src/postgresql-7.3.4/contrib/kdd/clasificacion
- Para compilar e instalar, se digitan las siguientes secuencias de comandos:
  # gmake
  # gmake install

**3. Creación de Usuario de Sistema**

Se ingresa como usuario root y se digita las siguientes sentencias de comandos:
# adduser <nuevo_usuario> (agrega nuevo usuario)
# passwd <password> (agrega password)
# usermod –g postgres <nuevo_usuario> (cambia <nuevo_usuario> a grupo postgres)

Ejemplo:
root@servidor:# adduser usuariokdd
root@servidor:# passwd kdd123456
root@servidor:# usermod –g postgres usuariokdd

**4. Creación de Base de Datos y Usuario de la Base de Datos**

Se ingresa como usuario postgres y se digita las siguientes sentencias de comandos:

- Se crea la base de datos a utilizar
  # su – postgres
  $ /usr/local/pgsql/bin/createdb <nueva_base_datos>

  Ejemplo:
  root@servidor:# su – postgres
  postgres@servidor:$ createdb prueba

- Se realiza la conexión a la base de datos y se crea el usuario a esta.
  $ /usr/local/pgsql/bin/psql <nueva_base_datos>
  =# CREATE USER <nuevo_usuario>;

  Ejemplo:
  postgres@servidor:$ psql prueba

prueba=# CREATE USER usuariokdd;

**5. Conexión como Nuevo Usuario**

Se digita la siguiente sentencia de comandos:
# /usr/local/pgsql/bin/psql <nueva_base_datos>

Ejemplo:
usuariokdd@servidor:$ psql prueba
Ahora esta conectado a la base da datos prueba.
prueba=>

**6. Creación de funciones de Asociación y Clasificación.**

Para asociación:

- Identificarse como usuario y conectarse a la base de datos a través de la terminal interactiva de postgresql.
  # su – <nombre_usuario>
  $ psql <nombre_base_datos>

  Ejemplo:
  usuariokdd@servidor:$ psql prueba

- Ejecutar el archivo inicio_tipo.sql que carga la función para crear el tipo de datos que utiliza describe_association_rules.
  => \i /usr/local/src/postgresql-7.3.4/contrib/kdd/asociación/inicio_tipo.sql
- Ejecutar la sentencia SQL
  => SELECT type_rules(‘<nombre_tabla_asociacion>’);

  Ejemplo:
  prueba=> SELECT type_rules('tabla_ejemplo_asso');
  *nota:* tabla_ejemplo_asso es una tabla resultado de la ejecución de las sentencias de Asociación

- Ejecutar el archivo inicio_funcion.sql que carga las funciones de generación de reglas de asociación.
  => \i /usr/local/src/postgresql-7.3.4/contrib/kdd/asociación/inicio_funcion.sql
- Ejecutar la sentencia SQL
  => SELECT * FROM describe_association_rules(‘<nombre_tabla_asociacion>’, tamaño_reglas, confianza);

  Ejemplo:

prueba=> SELECT * FROM describe_association_rules('tabla_ejemplo_asso', 3, 50);

Para Clasificación:

- Identificarse como usuario y conectarse a la base de datos a través de la terminal interactiva de postgresql.
  # su – <nombre_usuario>
  $ psql <nombre_base_datos>

  Ejemplo:
  usuariokdd@servidor:$ psql prueba

- Se ejecuta el archivo 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>');

  Ejemplo:
  prueba=> SELECT * FROM entro('tabla_ejemplo_class');
  *nota:* tabla_ejemplo_class es una tabla resultado de la ejecución de las sentencias de Clasificación.

- 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();