RNPS No. 2143
ISSN 2072-6260
Versión Digital

SERIE GRIS

REPORTE TÉCNICO
Minería de Datos

Algoritmos paralelos para el descubrimiento de secuencias frecuentes, cerradas y maximales

Ing. José Carlos Almagro Rosell,
Dr. C. José Hernández Palancar

RT_013 **abril 2010**

CENATAV
Centro de Aplicaciones de Tecnologías de Avanzada
MINISTERIO DE LA INDUSTRIA BÁSICA

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
Ciudad de La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

RNPS No. 2143
ISSN 2072-6260
Versión Digital

SERIE GRIS

REPORTE TÉCNICO
Minería de Datos

Algoritmos paralelos para el descubrimiento de secuencias frecuentes, cerradas y maximales

Ing. José Carlos Almagro Rosell,
Dr. C. José Hernández Palancar

RT_013 **abril 2010**

CENATAV
Centro de Aplicaciones de Tecnologías de Avanzada
MINISTERIO DE LA INDUSTRIA BÁSICA

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
Ciudad de La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

Índice

Algoritmos paralelos para el descubrimiento de secuencias frecuentes, cerradas y maximales

Ing. José Carlos Almagro Rosell, Dr. C. José Hernández Palancar

Centro de Aplicaciones de Tecnología de Avanzada, 7a #21812 e/ 218 y 222, Siboney, Playa, Ciudad de La Habana, Cuba
jalmagro@cenatav.co.cu

RT_013 CENATAV
Fecha del camera ready: 26 de marzo de 2010

Resumen: El minado de secuencias frecuentes es un importante problema de minería de datos, con amplias aplicaciones en varias áreas. Sin embargo los grandes volúmenes de datos a procesar y el gran número de patrones secuenciales que se genera a menudo, producto del avance tecnológico de los últimos tiempos, son dos factores que atentan contra la eficiencia del minado de secuencias frecuentes. En muchas aplicaciones el minado de todas las secuencias frecuentes no es necesario y el minado de Secuencias Frecuentes Cerradas y Maximales aportan la misma cantidad de información. En este reporte técnico se presenta un estado del arte sobre las principales técnicas para el Minado de Secuencias Frecuentes Cerradas y Maximales así como de sus correspondientes Algoritmos Paralelos, implementados con técnicas y herramientas de procesamiento paralelo, que permiten resolver el problema de la Escalabilidad ante grandes volúmenes de datos.

Palabras clave: minería de datos, secuencias frecuentes cerradas y maximales, escalabilidad, algoritmos paralelos

Abstract: The mining of frequent sequences is an important data mining problem with wide applications. However, huge volumes of data to be processed and the large number of sequential patterns are often generated, resulting from technological advances of recent years, are two factors that undermine the efficiency of the mining of sequences frequent. In many applications, the mining of all frequent sequences is not necessary and the mining of Frequent Closed and Max Sequences offers the same amount of information. This technical report presents a state of the art on the main techniques for Frequent Closed and Max Sequence Mining and their corresponding parallel implementations, with techniques and tools parallel processing, which allow solving the problem of scalability to large volumes of data.

Keywords: Data Mining, Frequent Closed Sequences and Maximal, Scalability, Parallel Algorithms

1 Introducción

En la actualidad se ha visto un acelerado crecimiento en cuanto a la disponibilidad de diversos tipos de datos en investigaciones científicas de gran impacto a nivel internacional, como el estudio de las secuencias de ADN y las proteínas, por poner un ejemplo, que contienen gran densidad de patrones. La Minería de datos ha contribuido considerablemente en el proceso de Descubrimiento de Conocimiento; muchas han sido las definiciones dadas a este proceso entre las que se encuentra:

- El proceso de extracción de información interesante (no trivial, implícita, previamente desconocida y potencialmente útil) de grandes base de datos u otro repositorio de información [23].

Entre todos los métodos de Minería de Datos existentes (predictivos y descriptivos), los más explotados a nivel mundial son *la clasificación* y *las reglas de asociación*, aunque en los últimos años se ha trabajado bastante en dirección a la minería de patrones frecuentes.

Con respecto a este último varios han sido los trabajos que abordan el minado de conjuntos de elementos o items frecuentes, y en menos proporción el minado de secuencias frecuentes. Como pudiera causar confusión la similaridad de estos términos (conjunto de items y secuencia de items) a continuación se presentarán sus principales diferencias:

- En un conjunto no se repiten los items y en una secuencia sí pueden repetirse.
- En un conjunto no importa el orden de los items y en una secuencia sí es importante el orden.

El presente trabajo aborda el minado de secuencias frecuentes (FS por sus siglas en inglés).

La minería de FS, juega un papel muy importante en la Minería de datos, con amplias aplicaciones en algunas áreas como el análisis del genoma humano (estudio de secuencias de ADN), análisis de secuencias de compra de clientes, análisis de patrones de acceso de consultas XML, etc. Para muchas de estas aplicaciones el minado de todas las FS no es necesario, ya que en ocasiones, genera una enorme cantidad de patrones secuenciales que no son de interés. Para evitar este fenómeno, varias literaturas introducen el enfoque de minado de Secuencias Frecuentes Cerradas y Maximales (FCMS por sus siglas en inglés), que genera un número menor de patrones con la misma cantidad de información, y por lo tanto mejora la eficiencia y la eficacia de estas tareas.

Aunque hoy día se han implementado varios algoritmos series para este tipo de minado, estos no ofrecen escalabilidad en términos de rendimiento, para grandes bases de datos, porque siempre existe un límite para el rendimiento de un único procesador (en cuanto a memoria y CPU). Entonces nos encontramos ante el desafío de buscar alguna alternativa (lógica y barata), que nos ofrezca la posibilidad de aumentar el volumen de datos a analizar, de manera más eficiente y así lograr desarrollar versiones escalables de los algoritmos comúnmente más utilizados [16]. Para lograr que un algoritmo de minería de datos sea escalable ante grandes volúmenes de datos, muchas aplicaciones y sistemas utilizan la computación de alto rendimiento (high performance computing) o computación paralela.

En un algoritmo escalable el tiempo de ejecución crece linealmente en proporción al tamaño de la base de datos, dados los recursos disponibles de memoria principal y espacio en disco. Los tres enfoques principales para aumentar la escalabilidad [8] son:

- El diseño de un algoritmo rápido: la mejora de la complejidad computacional, optimizando la búsqueda y la representación de soluciones aproximadas a los complejos problemas computacionales, o aprovechar el paralelismo inherente de la tarea.
- Particionar los datos: dividir los datos en subconjuntos (basado en casos o características), el aprendizaje de uno o más de los subconjuntos seleccionados, y la posible combinación de los resultados.
- Usando una representación relacional: las direcciones de datos que no son factibles pueden ser tratadas como un único archivo plano.

El reporte consta de 6 epígrafes. En el epígrafe 2 se presentan algunos conceptos básicos necesarios a lo largo del documento. En el epígrafe 3 se aborda el trabajo relacionado tanto para el minado de FS como para el minado de FCMS. En el epígrafe 4 se presentan los conceptos más utilizados dentro de la computación paralela. En el epígrafe 5 se describen los principales algoritmos paralelos implementados para el minado de FCMS y en el epígrafe 6 se dan las conclusiones, así como el trabajo futuro.

2 Conceptos básicos

El minado de secuencias frecuentes es una de las tareas más investigadas en el campo de la minería de datos. Desde que Agrawal y Srikant [3] introdujeron su magnífico enfoque sobre el minado de secuencias, muchos investigadores han sido capaces de utilizarlo como medio para resolver múltiples problemas en el campo de minería de datos. A continuación

introduciremos la terminología y las definiciones necesarias para establecer el problema de la Minería de Secuencias frecuentes, a partir de grandes conjuntos de datos.

Secuencia y conjuntos de items

Una secuencia S es una lista ordenada de elementos o itemsets (también llamados eventos por algunos autores) y es denotada como:

S =<S1 S2…Sn>, donde Sj es un itemset, tal que $1 \leq j \leq n$.

Un elemento de la secuencia es denotado por (x1, x2,…, xm) donde xj es un item. Un item puede ocurrir solamente una vez en un elemento de una secuencia, pero puede ocurrir múltiples veces en diferentes elementos.

Un itemset es considerado una secuencia con un solo elemento. Se asume además que los items en un elemento de una secuencia están en orden lexicográfico.

Subsecuencias y supersecuencias

Una secuencia <a1 a2…an> es una subsecuencia de otra secuencia <b1 b2…bm> si existen enteros < i1<i2 <…<intal que: a1$\subseteq$ bi1, a2$\subseteq$ bi2,…, an $\subseteq$ bin.

Por ejemplo la secuencia < (2) (3, 5) (9)> es una subsecuencia de < (7) (2, 4) (8) (3, 5,7) (9)>, ya que (2)$\subseteq$ (2, 4), (3, 5) $\subseteq$ (3, 5, 7) y (9) $\subseteq$ (9). Sin embargo la secuencia < (4) (7) > no es una subsecuencia de < (4, 7) > y viceversa.

Base de datos de secuencias

Una base de datos de secuencias, denotada como D = {S1, S2,…, Sn}, está formado por un conjunto de secuencias de entradas, donde cada secuencia está asociada a un único identificador (sid), y cada itemset en una secuencia tiene también un único identificador (eid). Se asume que cada secuencia no presenta más de un itemset con el mismo eid, (ver Tabla 1).

Tabla 1. Base de Datos

SID	EID	ítems
10	5	a, c
10	10	d
10	20	a
10	25	b, d
20	5	a, e
20	15	d, f
20	20	a
20	25	e, h
30	10	d, g
30	15	a, e
30	20	b
30	25	a
30	30	e
40	15	a, f
40	20	d, f
40	25	c
50	5	b
50	10	a, c
50	15	a
50	20	d
50	25	a
50	30	b, c, f

Tamaño de una secuencia

El tamaño (*size*), *n*, de una secuencia es el número de itemsets en la secuencia y se denota como |S|.

Longitud de una secuencia

La longitud (*length*), *l*, de una secuencia *S* se define como el número de items en la secuencia, ver (1). Una secuencia de longitud *k* es denotada como *k*-secuencia.

$$l = \sum_{i=1}^{n} |S_i| \tag{1}$$

Secuencia prefijo

Dadas dos secuencias $\alpha = \langle a_1 a_{2...} a_n \rangle$ (donde cada a_i corresponde a un elemento frecuente en S) y $\beta = \langle b_1 b_2 \ldots b_m \rangle$ ($m \leq n$). Se dice que β es un prefijo de α, si y solo si 1) $b_i = a_i$ para ($i \leq$ m-1); 2) $b_m \subseteq a_n$; y 3) todos los items frecuentes en ($a_n - b_m$) están alfabéticamente después de las de b_m.

Ej.1- Dada la secuencia α =⟨a(abc)(ac)d(cf)⟩; la secuencia β =⟨a(abc)a⟩ es un prefijo con respecto a α.

Proyección de secuencias

Dadas dos secuencias α y β, tal que β es una subsecuencia de α. Una subsecuencia α´ de la secuencia α, es llamada una proyección de α con respecto al prefijo β si y solo si: 1) α´ tiene prefijo β; 2) no existe una súper-secuencia α´´ de α´, tal que α´´ es una subsecuencia de α y también tiene prefijo β.

Ej.2- Dada la secuencia α =⟨a(abc)(ac)d(cf)⟩ con prefijo β = ⟨(bc)a⟩; entonces la subsecuencia α´= ⟨(bc)(ac)d(cf)⟩, es la proyección de α con respecto al prefijo β.

Secuencia sufijo

Dada una secuencia $\alpha = \langle a_1 a_{2...} a_n \rangle$ (donde cada a_i corresponde a un elemento frecuente en S). Sea β = β = $\langle b_1 b_2 \ldots b_m \rangle$ ($m \leq n$) el prefijo de α. Una secuencia $\gamma = \langle a_m'' a_{m+1} \ldots a_n \rangle$ es llamada sufijo de α con respecto al prefijo β, y es denotada como γ = α/β, donde $a_m'' = (a_m - a_m')$.

Ej.3- Sea α´=⟨a(abc)(ac)d(cf)⟩ una proyección de α con respecto al prefijoβ =⟨a(abc)a⟩, entonces γ =⟨(_c)d(cf)⟩ es el sufijo de α con respecto al prefijo β.

Soporte de una secuencia

El soporte de una secuencia α en una base de datos *D*, se define como el número de secuencias en *D* que contienen α. En otras palabras es la frecuencia de aparición de determinadas secuencias en la base de datos.

$$Soporte(\alpha) = \left| \{ S | S \in D \} \alpha \subseteq S \right| \tag{2}$$

Secuencias frecuentes

Dado un umbral especificado por el usuario, llamado soporte mínimo (denotado por minsup) se dice que una secuencia es frecuente (FS) si el soporte es mayor que el minsup.

$$Soporte(\alpha) \geq \min\sup \tag{3}$$

El concepto de soporte es bien utilizado cuando se habla de Minado de Secuencias Frecuentes. Agrawal formuló en [2] la propiedad de anti- monotonicidad, la cual plantea que toda subsecuencia/subconjunto de una secuencia/itemset frecuente, debe ser frecuente. Esta propiedad permite podar el espacio de búsqueda evitando calcular el soporte de conjuntos de secuencias que no pueden ser frecuentes.

Secuencias frecuentes maximales

Una secuencia *S* es llamada Secuencia Frecuente Maximal (FMS) en una base de datos D de secuencia, si *S* es una secuencia frecuente en D y no existe una secuencia frecuente *S1*, tal que *S1* sea una supersecuencia de *S.*

Secuencias frecuentes cerradas

Una secuencia S es llamada Secuencia Frecuente Cerrada (FCS) en una base de datos D de secuencias, si S es una secuencia frecuente en D y no existen secuencias frecuentes S1 tal que:

1. *S1* sea una supersecuencia de *S*,
2. Soporte(*S*) = Soporte(*S1)*

Tabla 2. Base de Datos de secuencias

Sid	Secuencias
10	<(ac)da(bd)>
20	<(ae)(df)a(eh)>
30	<(bg)(ae)bae>
40	<(af)(df)c>
50	<b(ac)ada(bcf)>

3 Algoritmos series para el minado de secuencias frecuentes

En el campo de la minería de datos, el descubrimiento de secuencias frecuentes ha sido estudiado desde 1995 [3, 4, 11, 9, 12, 21]. La mayoría de los algoritmos implementados para el minado de secuencias frecuentes, utilizan tres tipos diferentes de enfoques de acuerdo a la forma de evaluar el soporte de los patrones secuenciales candidatos. En la Figura 1 se muestra una taxonomía de los principales algoritmos.

1. El primer grupo de algoritmos se basan en la propiedad Apriori, introducida por Agrawal and Srikant [2] en el minado de reglas de asociación. Esta propiedad plantea que cualquier subpatrón de un patrón frecuente también es frecuente, permitiendo podar las secuencias candidatas durante el proceso de generación de candidatos. Basado sobre esta heurística se propusieron algoritmos como el AprioriAll y el AprioriSome en [3]. En trabajos posteriores (1996) los mismos autores proponen el algoritmo GSP (Patrón secuencial generalizado) [4] basado también en técnicas Apriori y superando a los anteriores en 20 magnitudes de tiempo. En estos algoritmos el mayor esfuerzo se centró en desarrollar estructuras específicas que permitieran representar los patrones secuenciales candidatos y de esta manera realizar las operaciones de conteo del soporte con mayor rapidez (ej. uso de tablas hash).
2. El segundo grupo lo forman algoritmos que tratan de reducir el tamaño del conjunto de datos explorados, mediante la realización de proyecciones de la base datos inicial y el crecimiento de patrones, sin generación de candidatos. Utilizando esta técnica y bajo la filosofía de divide y vencerás, Pei y Han proponen el algoritmo FreeSpan (Frecuent Pattern-Project Sequential Pattern mining) [9] y el PrefixSpan (Prefix-projected Sequential Pattern mining) [12]. En estos algoritmos la base de datos de secuencia es proyectada recursivamente en un conjunto de pequeñas bases de datos, basado en el conjunto actual de Secuencias Frecuentes, y minan cada base de datos proyectada para encontrar sus patrones.

3. El tercer grupo está formado por algoritmos que mantienen en memoria solamente la información necesaria para la evaluación del soporte. Estos algoritmos se basan en las llamadas listas de ocurrencia que contienen la descripción de la ubicación donde ocurren los patrones en la base de datos. Bajo este enfoque, en el 2001 Mohammed Zaki propone el algoritmo SPADE (Sequential Pattern Discovery using Equivalence classes) [11] donde introduce la técnica de transformación de la base de datos a formato vertical.

Un novedoso algoritmo basado en la indexación de memoria para el rápido descubrimiento de patrones secuenciales llamado MEMISP (MEMory Indexing for Sequential Pattern mining) [20] fue introducido en el 2005. Con MEMISP no hay generación de candidatos ni proyección de bases de datos.

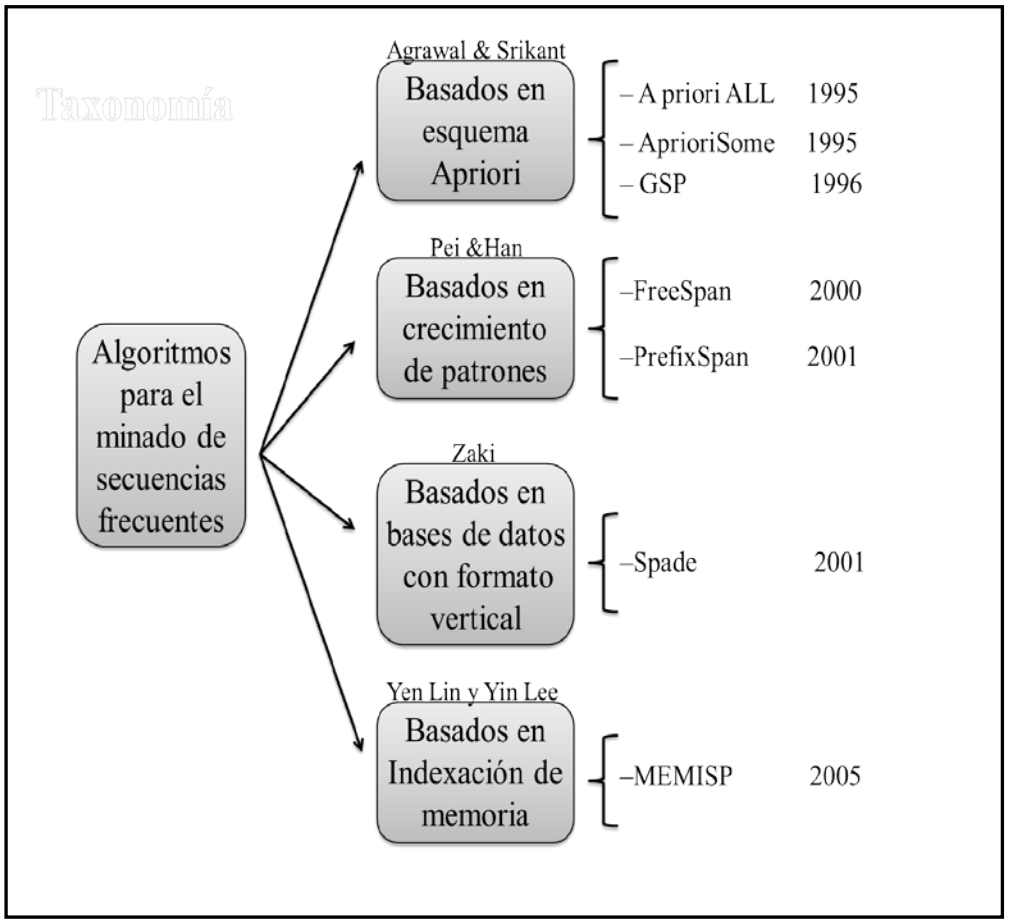

Fig. 1. Taxonomía de los principales algoritmos para el minado de secuencias frecuentes

3.1 Algoritmos basados en Apriori

El algoritmo AprioriAll presentado por Agrawal and Srikant en [3] fue el primer método utilizado para el minado de secuencias frecuentes, basado sobre el algoritmo Apriori en el minado de reglas de asociación. Dada una base de datos de transacciones con tres atributos (similar a la que se muestra en la Tabla 1), los autores en [3], dividen el proceso de minado en 5 fases: 1) Fase de Ordenamiento 2) Fase de Litemset, 3) Fase de Transformación ,4) Fase de Secuencias y 5) Fase de determinación de Máximos.

Fase de Ordenamiento: Ordena la base de datos con el id de clientes como llave principal y el tiempo de transacción como llave secundaria (ver tabla 1). En esta fase se construyen las secuencias de clientes (ver tabla 2).

Fase de Litemset: En esta fase se busca el conjunto de todos los litemsetsL: conjunto de todas las 1-secuencias frecuentes. Se puede utilizar cualquier algoritmo para buscar itemsets frecuentes en el contexto de reglas de asociación.

Fase de transformación: Se construye una nueva base de datos que contiene los litemsets encontrados en cada transacción, bajo la idea de que si un itemset i no es frecuente, entonces no existe una secuencia frecuente que contenga a i.

Fase de Secuencia: Usa el conjunto de litemsets para encontrar las secuencias deseadas. Usando un conjunto primario de litemsets, se generan las nuevas FS, llamadas Secuencias Candidatas y denotadas por Ck. Durante los pases por la base de datos es calculado el soporte, y en el último pase se determina cual secuencia es aún un litemsets.

Fase Maximal: Busca las secuencias maximales entre todo el conjunto de secuencias frecuentes.

Este algoritmo presenta generación de candidatos y mina todos los patrones sin tener en cuenta la existencia de restricciones de espacio. Se considera a una FS, si todos sus elementos están presentes (en el orden dado), en un número suficiente de secuencias en la base de datos. La generación de candidatos en este caso se realiza por la unión de dos (k-1)-secuencias frecuentes cuando sus prefijos máximos son iguales.

Otro de los algoritmos presentados en [3] es el AprioriSome el cual parte de la idea de que, como solo interesan secuencias máximas, se puede evitar contar secuencias que estén contenidas en otras más grandes, que ya han sido contadas. Sin embargo hay que tener cuidado de no contar las secuencias más largas que no cumplan con el soporte mínimo.

Estos algoritmos son comparables en cuanto a su rendimiento, aunque AprioriSome trabaja un poco mejor para valores más bajos de soporte mínimo.

La gran ventaja de estos algoritmos reside en su carácter iterativo. Sin embargo cuando son utilizadas restricciones de espacio, estos algoritmos presentan limitaciones en cuanto a la generación de todos los candidatos. Esta propiedad condujo, eventualmente a la definición de un nuevo algoritmo basado también en el esquema Apriori, pero que no presenta esta limitante, llamado GSP.

3.1.1 GSP

El Algoritmo GSP (Patrón Secuencial Generalizado) [4] presentado también por Agrawal and Srikant (1996), a diferencia del AprioriAll y AprioriSome, incorpora en el proceso de minado, algunas restricciones de espacio como las limitantes de tiempo, donde los autores definen el gap máximo y gap mínimo, para especificar el espacio entre dos transacciones adyacentes en una secuencia. Si la distancia entre dos transacciones no se encuentra en el rango del gap, entonces estas transacciones no son tomadas como consecutivas en una secuencia. También introduce la ventana de tiempo (sliding windows); si el rango entre el tiempo máximo y mínimo de una transacción es menor que el sliding windows, estos items pueden ser considerados que se encuentran en una misma transacción. El algoritmo además introduce el término de taxonomía para generar patrones secuenciales en múltiples niveles.

El algoritmo GSP realiza múltiples pases sobre la Base de Datos. En el primer pase todos los items simples (1-secuencias) son contados para encontrar cuales son frecuentes. A partir de los items frecuentes, un conjunto de *2*-secuencias candidatas es formado y su soporte es contado en el otro pase por la base de datos. A partir de las*2*-secuencias frecuentes son generadas las *3*-secuencias candidatas y este proceso es repetido hasta que no se encuentren nuevas secuencias frecuentes. El Algoritmo GSP consta de dos pasos para su implementación.

1. Generación de candidatos: Dado un conjunto de *(k-1)*-secuencias frecuentes (F_{k-1}), los candidatos, para el próximo nivel, son generados por la unión (*join*) de F_{k-1} consigo mismo. Una fase de poda elimina cualquier secuencia donde al menos una de sus subsecuencia no sea frecuente. Para un conteo más rápido, las secuencias candidatas son almacenadas en un árbol hash.

2. Conteo del soporte: Para encontrar todos los candidatos contenidos en una secuencia de entrada β, conceptualmente todas las k-subsecuencias son generadas. Si un candidato en el *árbol hash* coincide con la subsecuencia, el contador de soporte es incrementado.

Algoritmo1: GSP
Input: Base de datos de secuencias y Soporte mínimo. **Output**: Secuencias frecuentes contenidas en la Base de datos. 1 F_1= {*1*-secuencias frecuentes}; 2 **for** ($k=2$; $F_{k-1} \neq 0$; $k= k+1$) **do** 3 C_k = Conjunto de k-secuencias candidatas; 4 **for all** secuencias de entrada E en la base de datos **do** 5 Incrementar contador de $\alpha \in C_k$ contenida en E 6 $F_k = \{\alpha \in C_k \mid \alpha.sop \geq minsup\}$; 7 Conjunto de todas las secuencias frecuentes = $\bigcup_k F_k$

El algoritmo GSP presenta dificultad para contar el soporte de las secuencias candidatas debido a la incorporación de las limitantes de tiempo. Entones los autores aplican dos fases: *forward phase* y *backward phase*, las cuales son repetidas hasta que todos los patrones sean encontrados. Dado un conjunto de secuencias candidatas C y una secuencia de datos d, durante *forward phase*, GSP encuentra elementos sucesivos de C en d, siempre y cuando la diferencia entre el tiempo final de un elemento y el tiempo de comienzo del elemento previo sea menor que el *gap máximo;* si la diferencia es mayor, cambia a la *backward phase,* donde GSPtrata de adelantar los elementos previos, hasta que la diferencia satisfaga la condición del *gap máximo.* El algoritmo combina ambas fases.

Deficiencias del algoritmo GSP

El algoritmo GSP, como los anteriores basados en esquema Apriori, presenta como desventaja la gran cantidad de secuencias candidatas que genera, unido a los múltiples pases que necesita realizar por la base de datos, lo cual resulta ineficiente para la minería de largos patrones secuenciales. También en este sentido, cuando todas las secuencias candidatas no caben en la memoria principal, el algoritmo solo genera tantas candidatas como quepan en la memoria, que son procesadas luego para obtener las FS. Estas FS son copiadas al disco y el proceso se repite hasta que todas las candidatas sean contadas. Con este análisis lo que hemos querido presentar es que el algoritmo además provoca gasto de I/O al disco.

3.2 Algoritmos basados en el crecimiento de patrones

En los algoritmos presentados anteriormente, basados en esquemas Apriori, la enorme cantidad de candidatos generados representa una limitante para la escalabilidad del sistema. En esta sesión presentaremos un nuevo enfoque para enfrentar el problema de la minería de secuencias: el método de Crecimiento de Patrones. La idea principal es evitar el paso de la generación de candidatos y centrar la búsqueda sobre porciones restringidas de la base de datos inicial. Basado en este enfoque, Han introduce el algoritmo FreeSpan (Frequent Pattern-projected Sequential Pattern mining) en [9]. FreeSpan, a partir de items frecuentes, proyecta la base de datos de secuencias recursivamente. El tamaño de las bases de datos proyectadas, a menudo, decrece rápidamente, y estas pequeñas bases de datos son más fáciles de manejar. Este método es considerablemente más rápido que los métodos basados en esquemas Apriori. El problema de este algoritmo es que la misma secuencia puede estar duplicada en varias bases de datos. En un trabajo posterior Pei introduce el PrefixSpan [12], que no solo elimina el problema de las secuencias duplicadas, sino que permite minar eficientemente grandes bases de datos de secuencias.

3.2.1 PrefixSpan

PrefixSpan [12] es el más prometedor de los algoritmos de crecimiento de patrones y se basa en la construcción recursiva de patrones. La gran ventaja de PrefixSpan es el uso de bases de datos proyectadas, para así lograr bases de datos mucho más pequeñas en el próximo nivel, que el algoritmo pueda procesar con mayor rapidez. El Algoritmo 2 muestra el pseudocódigo para la implementación del PrefixSpan.

Una base de datos proyectada α es el conjunto de subsecuencias en la base de datos, que son sufijos de una secuencia α con prefijo β; se denota como $S|\alpha$.

El minado de secuencias frecuentes a partir del algoritmo PrefixSpan se divide en 3 pasos fundamentales:

1. Búsqueda de *1*-secuencias frecuentes.
2. Dividir el espacio de búsqueda, en diferentes particiones de acuerdo al número de *1*-secuencias frecuentes (Figura 2). Cada partición es la proyección de la base de datos de secuencias, que tenga la correspondiente *1*-secuencia frecuente como prefijo. La base de datos proyectada sólo tiene los sufijos de la secuencia correspondiente.
3. Búsqueda de subconjuntos de patrones secuenciales. Se realiza mediante la construcción de las correspondientes bases de datos proyectadas y la minería de cada una, de forma recursiva.

Este algoritmo resulta más eficiente que los presentados anteriormente ya que no necesita generar candidatos frecuentes; sin embargo la construcción de bases de datos proyectadas es el mayor costo, en cuanto a tiempo y espacio, de este algoritmo. Para mejorar esta característica existen dos técnicas diferentes de proyección: la *proyección bi-nivel* y la *pseudo- proyección* que reducen el costo de la proyección sustancialmente.

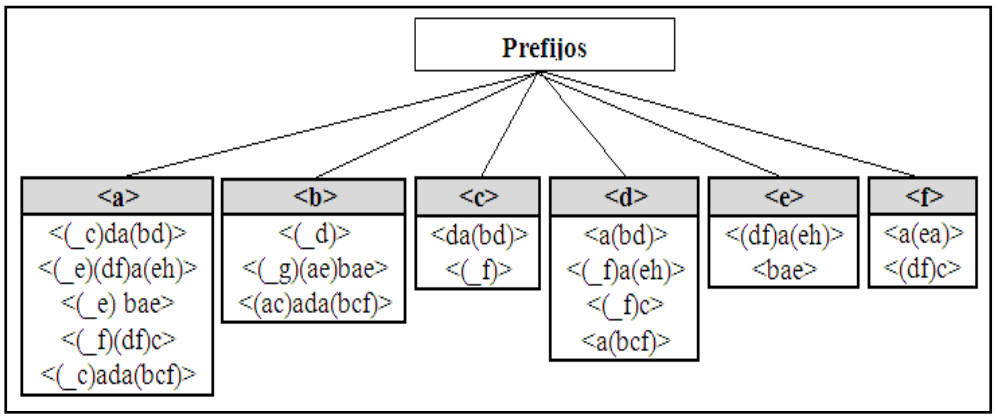

Fig. 2. Base de datos proyectada

Proyección bi- nivel

Cuando la base de datos proyectada no se puede mantener en la memoria principal, y por tanto no puede utilizarse la pseudo-proyección, se aplica entonces la *proyección bi-nivel* para reducir el número y tamaño de las bases de datos proyectadas. El primer paso para esta proyección es el mismo: se obtienen todas las *1*-secuencias frecuentes; en el segundo paso en vez de construir una base de datos proyectada, se construye una matriz triangular n x n, que registra el soporte de todas las 2-secuencias formadas a partir de *1*-secuencias frecuentes. Para cada *2*-secuencia frecuente α se construye una base de datos proyectada α y se buscan los items frecuentes a partir de los cuales se construye su correspondiente matriz. Estos pasos se repiten hasta que la base de datos proyectada esté vacía o no se encuentren nuevas secuencias frecuentes. Con la utilización de la matriz triangular, el número de bases de datos proyectadas es más pequeño y por tanto se requiere menos espacio.

Algoritmo 2: PrefixSpan

Input: Base de datos de secuencias S y soporte mínimo.
Output: Conjunto completo de secuencias frecuentes.

Método: **call** PrefixSpan*(<>, 0,S)*

Subrutina PrefixSpan(α, l, $S|_\alpha$**)**
//Parámetros: α: secuencia; l: longitud de α; $S|_\alpha$: base de datos proyectada α,
//si $\alpha \neq <>$, de lo contrario será la base de datos de secuencias S.

Método:

1 Explora $S|_\alpha$ una vez y encuentra el conjunto de items frecuentes b, tal que:
b pueda ser añadido al último elemento de α para formar una secuencia;
ó $<b>$ pueda ser anexado a α para formar una secuencia.
2 **for each** item frecuente b,
anexar b a α para obtener una secuencia α';
3 **for each** α',
construir una base de datos proyectada $S|_\alpha$;
call PrefixSpan(α', l+1,$S|_\alpha$);

Pseudo- proyección

La pseudo-proyección se utiliza cuando una base de datos proyectada cabe en la memoria principal. Sin la necesidad de realizar una proyección física, la pseudo-proyección es más eficiente que otros métodos de proyección, evitando copiar la base de datos. Cada proyección consta de dos elementos de información: *punteros* referidos a la secuencia en la base de datos y el *offset* del sufijo en la secuencia.

Ventajas del PrefixSpan

El algoritmo PrefixSpan es más eficiente que los algoritmos basados en esquemas Apriori, presentados anteriormente. Bajo el método de crecimiento de patrones, este algoritmo realiza una búsqueda más centrada sobre los subespacios que aporten una mayor cantidad de patrones y evita la generación de candidatos, que veíamos como limitante en el algoritmo GSP. Su consumo de memoria es relativamente estable. El principal problema del PrefixSpan es el tiempo que consume en explorar las bases de datos proyectadas, que puede ser muy grande si el conjunto de datos original es enorme.

3.3 Algoritmos con base de datos en formato vertical: SPADE

El Algoritmo SPADE (Sequential Pattern Discovery using Equivalence classes) propuesto por Zaki en [11], a diferencia que los algoritmos basados en Apriori, no realiza múltiples pases sobre la base de datos, y pude minar todas las secuencias frecuentes en solo tres pases. Esto se debe a la incorporación de nuevas técnicas y conceptos entre las que se encuentran las siguientes:

- Usa una lista de identificadores (id-list) con formato vertical (ver Figura 2), donde se asocia a cada secuencia con una lista de identificadores (SID), junto con el identificador de evento temporal (TID). Sobre esta lista pueden ser enumeradas todas las secuencias frecuentes mediante simples uniones temporales.
- Utiliza un enfoque de retículo o *lattice*[1], para descomponer el espacio de búsqueda original en clases pequeñas (sub-lattice), que pueden ser procesadas independientemente en la memoria principal (ver Figura 3).
- Propone dos estrategias de búsqueda diferentes para enumerar las secuencias frecuentes dentro de cada clase: Búsqueda en amplitud y búsqueda en profundidad.

Tabla 3. Id-list con formato Vertical para los items a, b, c, d, e, f

a		b		c		d		e		f	
SID	TID	SID	TID	SID	TID	SID	TID	SID	TID	SID	TID
10	1	10	4	10	1	10	2	20	1	20	2
10	3	30	1	40	3	10	4	20	4	40	1
20	1	30	3	50	2	20	2	30	2	40	2
20	3	50	1	50	6	40	2	30	5	50	6
30	2	50	6			50	4				
30	4										
40	1										
50	2										
50	3										
50	5										

Para calcular todas las 1-secuencias frecuentes en una base de datos con formato vertical basta con realizar un simple pase por la base de datos, e incrementar el soporte para cada nuevo *sid* encontrado (por ejemplo el item *a* presenta soporte igual 5).

Para el cálculo de las 2-secuencias frecuentes, la cantidad de datos que se requiere leer desde la id-list en la base de datos es muy grande. Para eliminar esto el algoritmo propone desarrollar una transformación de la base de datos, de formato Vertical a Horizontal.

Conteo del soporte

En los algoritmos presentados anteriormente se asume un diseño de base de datos horizontales como el de la Tabla 1. A diferencia este algoritmo utiliza un diseño de base de datos vertical, donde se asocia cada item de la secuencia a su id-list.

Dada una id-list de secuencia se puede determinar el soporte de cualquier k-secuencia por la simple intercepción de las id-list de dos de sus subsecuencias de longitud (k-1). En particular el SPADE propone dos subsecuencias de longitud (k-1) que compartan un prefijo común, para calcular el soporte de una nueva secuencia de longitud k. Un simple chequeo de la cardinalidad de la id-list resultante (el número de SID distintos), nos dice si una secuencia es frecuente o no.

Descomposición del retículo

Con las *2*-secuencias frecuentes generadas se construye un retículo, el cual es demasiado grande para que quepa en la memoria principal por lo que el SPADE descompone el retículo original en pequeñas clases, tal que cada clase pueda ser procesada independientemente en la memoria. Las secuencias que tienen un mismo item como prefijo pertenecen a la misma clase. En algunos casos, una clase aún puede ser demasiado grande para ser procesada en la memoria principal. En este caso, se aplica la descomposición de clase recursiva, vea la Figura 3.

En una tercera exploración por la base de datos todas las secuencias largas son enumeradas usando uniones temporales (ver Figura 4).

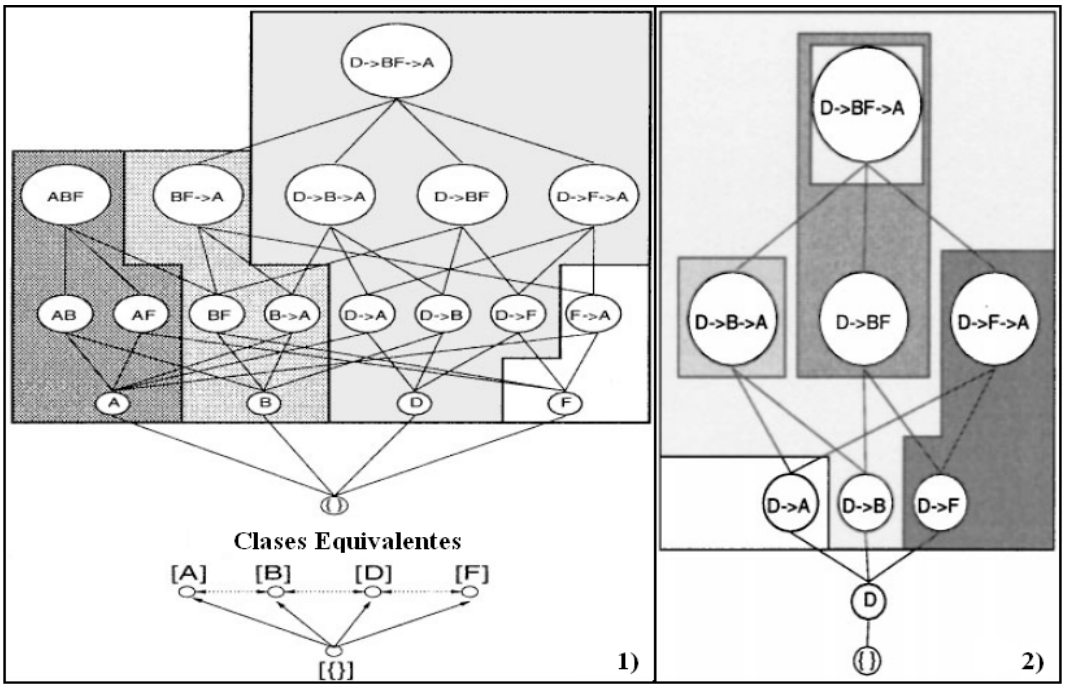

Fig. 3. 1) Descomposición del retículo, 2) Descomposición recursiva

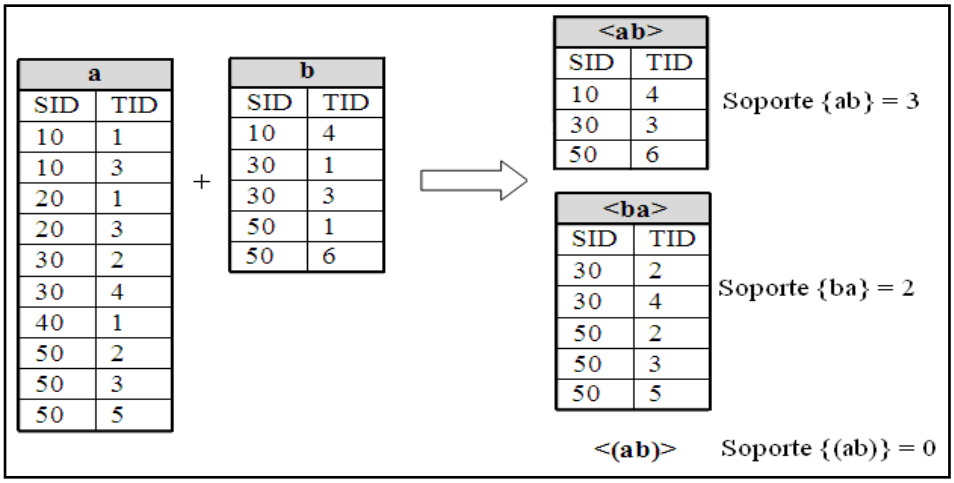

Fig. 4. Uniones temporales

Búsqueda de secuencias frecuentes

El SPADE propone dos estrategias de búsqueda eficiente para enumerar las secuencias frecuentes dentro de cada clase: La búsqueda en amplitud (*BFS* por sus siglas en inglés) y búsqueda en profundidad (*DFS*). Ambos métodos están basados sobre la descomposición recursiva de cada clase padre en pequeñas clases. En BSF el retículo de clases equivalentes es explorado recursivamente de manera ascendente, procesándose cada clase de un nivel antes de pasar al próximo. Por ejemplo para generar las *3*-secuencias frecuentes, todas las *2*-secuencias frecuentes tienen que haber sido procesadas. En DFS se procesan todas las clases hijas equivalentes a lo largo de una rama del retículo antes de pasar a la siguiente rama. Aunque BSF necesita más memoria principal, para almacenar todas las *2*-secuencias frecuentes, esta estrategia de búsqueda aporta más información para podar las *k*-secuencias candidatas.

En el Algoritmo 2 se muestra el pseudocódigo para una búsqueda en amplitud para el SPADE.

Algoritmo 3: SPADE

Input: Base de datos de secuencias y Soporte mínimo.
Output: Secuencias frecuentes contenidas en la Base de datos.

```
1   F1= {1-secuencias frecuentes};
2   F2={2-secuencias frecuentes};
3   C= {clases padres Ci= [Xi]}
4   for cada Ci∈C do  Enumerate-Frequent-Seq( Ci )

// PrevL es la lista de clases frecuentes del nivel previo
// NewLes la lista de las nuevas clases frecuentes para el nivel actual

5   Enumerate-Frequent-Seq(Prevl):
6   for (; PrevL ≠ 0; PrevL = PrevL.next())
7   NewL = NewL∪ Get-New-Classes(PrevL.item());
8   If (NewL ≠ 0) then Enumerate-Frequent-Seq( NewL);

9   Get-New-Classes(S):
10  for all  secuencias Ai∈S do
11  for all secuencias Ai∈S con  j ≥ i do
12  R = Ai ∪ Aj ;
13  L(R) = L(Ai ) ∩L(Aj );
14  if  (σ(R) ≥ minsup) then Ci= Ci∪{R};
15   CList= CList ∪Ci;
16  return CList;
```

El algoritmo SPADE supera al GSP, al introducir importantes optimizaciones que reducen el consumo de memoria y mejoran su eficiencia, como la utilización de las clases de equivalencia de secuencias frecuentes, las cuales pueden ser procesadas independientemente en la memoria principal utilizando eficientes técnicas de búsqueda sobre el retículo y simples uniones temporales. Todas las secuencias frecuentes son encontradas con solamente tres pases sobre la base de datos, y esto reduce el costo de I/O.

Sin embargo el SPADE se basa en listas que contienen información sobre la localización de los patrones en las secuencias y las repeticiones consecutivas conllevan a un desfavorable crecimiento del tamaño de estas listas de ocurrencia y por lo tanto aumenta el tiempo total de extracción, afectando de esta manera el rendimiento del algoritmo.

3.4 Algoritmos basados en la indexación de memoria

Con el fin de aumentar el rendimiento de la minería de datos, la utilización de la memoria debe ser aumentada sustancialmente para, de esta manera, reducir al mínimo el número de las operaciones sobre disco, especialmente cuando se trata de enormes bases de datos de secuencia. Basados en este reto fue introducido un novedoso algoritmo basado en la indexación de memoria para el rápido descubrimiento de patrones secuenciales llamado MEMISP.

3.4.1 MEMISP

El algoritmo MEMISP (MEMory Indexing for Sequential Pattern mining) presentado por Yen Lin y Yin Lee en [20], solamente requiere un pase sobre la base de datos, a lo sumo dos para bases de datos demasiado grandes; además evita la generación de candidatos y la proyección de base de datos, y tanto la utilización de la CPU como de la memoria, son altos.

MEMISP utiliza una estrategia de búsqueda e indexación para encontrar todas las secuencias frecuentes a partir de una secuencia de datos en la memoria. Primero lee toda la secuencia de datos en memoria y cuenta los soportes de las *1*-secuencias. Luego crea un conjunto de índices y entonces las secuencias frecuentes son encontradas utilizando las secuencias de datos indicadas por el conjunto de índices. El algoritmo 4 muestra el pseudo-código para la implementación de estos pasos.

Algoritmo 4 MEMISP

Input: *DB* = Base de datos de secuencias; *minsup* = soporte mínimo.
Output: conjunto de todas los patrones secuenciales.
Método:

1 Explorar *DB* en la memoria (*MDB*), y encontrar el conjunto de todos los items frecuentes.
2 **for** cada item frecuente *x*
   (i) formar y obtener el patrón secuencial $\rho = <(x)>$.
   (ii) **call** *IndexSet(x, <>, MDB*) para construir el conjunto de índices ρ*-idx.*
   (iii) **call** *Mine(*ρ*,* ρ*-idx)* para minar patrones con ρ*-idx.*

Subrutina IndexSet*(x,* ρ*, range-set)*
//Parámetros: *x*: stem-item; ρ =patrón*(P-pat); range-set* =conjunto de secuencias de datos para indexar.
//Output: conjunto de índices ρ*'-idx,* donde ρ*'* denota el patrón formado por *x* y *P-pat*
Método:

1 **for** cada secuencia de datos *ds* en *range-idx*
   (i) **if** *range-set* = *MDB* **then** *start-pos* = 0; **else** *start-pos* = *pos.*
   (ii) comenzar desde la posición (*start-pos + 1*) en *ds,*
   if el *stem-item x* se encontró por primera vez en la posición *pos* en *ds*
   then insertar un par *(ptr_ds, pos)* en ρ*'-idx,* donde *ptr_ds* apunta a *ds.*
2 **return** ρ*'-idx.*

Subrutina Mine*(*ρ*,* ρ*-idx)*
Método:

1 **for** cada secuencia de datos *ds* apuntado por *ptr_ds* de una entrada *(ptr_ds)* en ρ*-idx*
   (i) comenzando desde la posición *(pose+1)* hasta $|ds|$ en *ds* incrementar, en uno, el contador de soporte de cada stem *x* potencial.
2 Encontrar el conjunto de stem x cuyo contador del soporte sea ≥ que *minsup*, para formar un patrón secuencial.
3 **for** cada stem *x*,
   (i) formar y obtener el patrón secuencial ρ*'* con *P-pat*ρ y stem *x.*
   (ii) **call** *IndexSet(x ,*ρ*,*ρ*-idx)* para construir el conjunto de índices ρ*'-idx.*
   (iii) **call** *Mine*(ρ*', *ρ*'-idx*) para minar patrones con el conjunto de índices ρ*'-idx.*

Manejo de bases de datos extra-grande mediante particiones

Con el aumento del tamaño de las memorias (cada vez mayor), muchas bases de datos, ahora se pueden meter en la memoria principal de los ordenadores sin dificultad. Sin embargo, algunas bases de datos pueden ser demasiado grandes para caber en la memoria. En este caso, el minado de patrones secuenciales se realiza mediante la técnica de Partición y Validación como se muestra en la Figura 5.

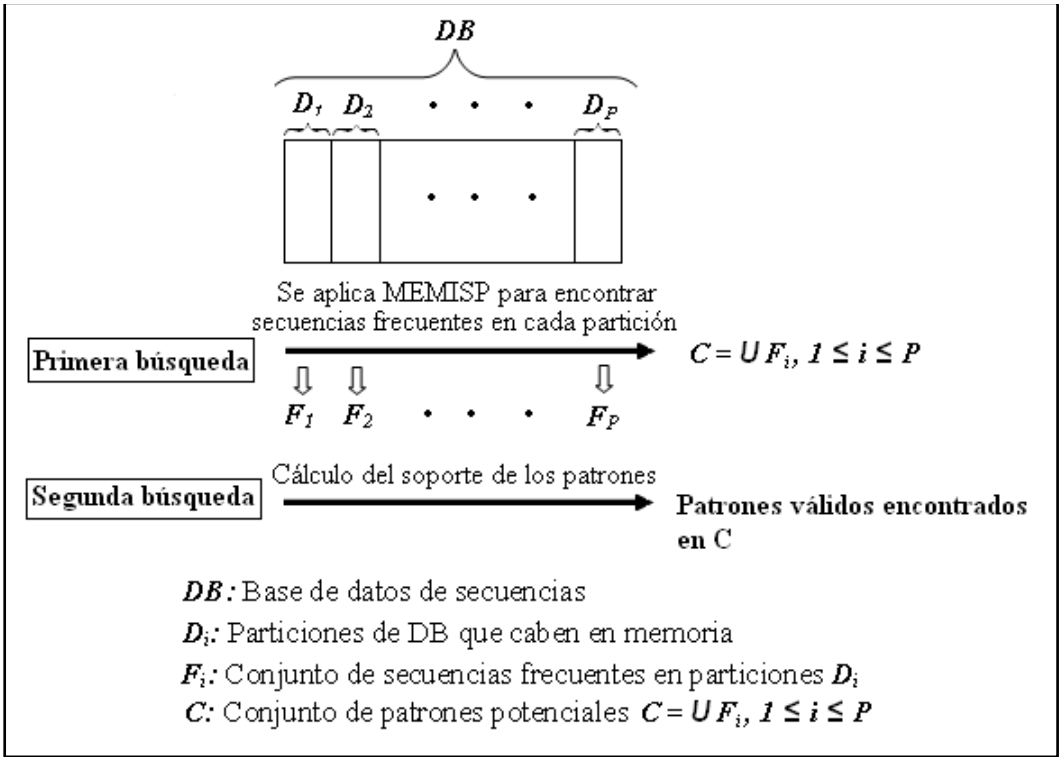

Fig. 5 Partición de la base de datos y descubrimiento de patrones para bases de datos extra largas

La base de datos extra larga es particionada de modo que cada partición pueda ser manejada en la memoria principal por MEMISP. El conjunto de patrones potenciales en la DB es encontrado mediante la recopilación de los patrones descubiertos después de correr MEMISP sobre estas particiones. Los patrones válidos pueden ser identificados con solamente una pasada por la base de datos extra larga, contando el soporte de todas las secuencias en la BD a la vez. Por lo tanto podemos emplear MEMISP para minar bases de datos de cualquier tamaño, con cualquier soporte mínimo, en solo dos pases de exploración sobre la base de datos.

Ventajas de MEMISP

En comparación con los otros algoritmos presentados anteriormente, MEMISP reduce el número total de pases sobre la base de datos a dos solamente, sin necesidad de espacio de almacenamiento adicional. GSP explora al menos *k* veces para descubrir *k*-secuencias frecuentes. SPADE necesita explorar la base de datos tres veces, además de espacio de almacenamiento en disco para la transformación de la base de datos vertical. PrefixSpan a menudo crea y procesa bases de datos proyectadas que son varias veces el tamaño de la base de datos original.

Resumiendo MEMISP es superior además porque:

- *No genera candidatos*: MEMISP descubre los patrones directamente desde la secuencia de datos en memoria mediante la utilización de la técnica de indexación.
- *No realiza proyección de bases de datos*: El simple y puro enfoque de indexación en MEMISP, no crea nuevas bases de datos, por lo que el almacenamiento intermedio, que requiere PrefixSpan, no es necesario aquí.

- *Se centra en la búsqueda e indexado efectivo*: MEMISP considera sólo las secuencias de datos indicadas por el conjunto de índices, en vez de buscar todas las secuencias de datos en la base de datos.
- *Compacto almacenamiento de índices*: MEMISP requiere poco espacio de almacenamiento para el conjunto de índices. En un conjunto de índices, el número máximo de índices necesario es igual al número de secuencias de datos, sin importar cuán pequeño sea el valor del *minsup*.
- *Alta utilización de memoria y la CPU*: MEMISP utiliza toda la memoria disponible y maximiza la utilización de la CPU, sin operaciones de disco adicionales.

4 Minado de secuencias frecuentes maximales y cerradas

En particular FS juega un papel importante en la Minería de Datos, sin embargo a menudo genera un gran número de patrones secuenciales, los cuales reducen la Eficiencia y la Eficacia de las tareas a ejecutar. Para reducir el enorme conjunto de patrones frecuentes generados, en estudios recientes muchos autores han estado trabajando sobre la compresión en el minado de estos patrones. Para muchas aplicaciones de la Minería de Datos, todos los patrones de las secuencias frecuentes no son necesarios; es así como se introduce el enfoque de minado de Secuencias Frecuentes Cerradas y Maximales (FCMS), que genera un número menor de patrones con la misma cantidad de información [17].

Los algoritmos para el minado de secuencias frecuentes han tenido un gran desarrollo en bases de datos que contienen secuencias frecuentes cortas. Pero desafortunadamente cuando se minan secuencias frecuentes largas, o cuando el umbral del soporte es muy bajo, el rendimiento de estos algoritmos decrece sustancialmente. Un problema similar ocurre en el minado de itemsets frecuentes. Una solución interesante, llamada *minado de itemsets frecuentes y cerrados* [6], ha sido propuesta para superar esta dificultad. En esta dirección varios algoritmos efectivos han sido desarrollados como el CLOSET [10], MAFIA [13], CHARM [15], y CLOSET + [21], para el minado eficiente de *itemsets frecuentes y cerrados*. Sin embargo, para nuestro conocimiento, menos han sido los algoritmos desarrollados para el minado de FCMS. En lo siguiente presentaremos algunos ejemplos de algoritmos utilizados para el minado de secuencias frecuentes cerradas (*FCS*): como el CloSpan y el Bide.

4.1 CloSpan

El algoritmo CloSpan (Closed Sequential Pattern mining) propuesto en [18] puede minar bases de datos de secuencias realmente largas, descubriendo un número menor de secuencias que los métodos tradicionales y aportando la misma cantidad de información. Desarrolla varios métodos eficientes para podar el espacio de búsqueda. CloSpan puede ser aplicado tanto para pequeñas como para grandes bases de datos: si la base de datos de secuencia cabe completamente en la memoria, que podría ser posible con la tecnología de hoy día, CloSpan puede ser aplicado directamente; en caso contrario la proyección basada en patrones frecuentes, puede ser aplicada antes de aplicar CloSpan.

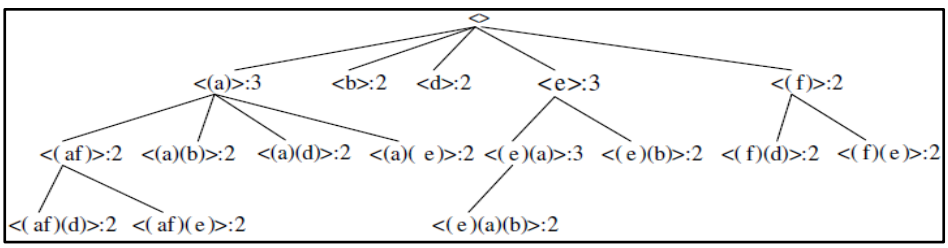

Fig. 6. Árbol de secuencias lexicográficas

CloSpan utiliza además el concepto de Árbol de Secuencia Lexicográficas, que es utilizado en bases de datos finitas para organizar todas las secuencias frecuentes en un árbol. En esta estructura se realiza una búsqueda en profundidad, para encontrar todas las FCS. En la Figura 6 se muestra un ejemplo de un árbol de secuencias lexicográficas.

Poda del espacio de búsqueda y retículo de secuencias de prefijos

CloSpan divide el proceso de minado en dos niveles. En el primer nivel es generado el conjunto de candidatos. Usualmente este conjunto de candidatos es mayor que el conjunto de secuencias cerradas final. El segundo nivel ayuda a eliminar las secuencias no cerradas.

Para podar el espacio de búsqueda, CloSpan introduce nuevas definiciones en el primer nivel, como: la *Equivalencia de bases de datos proyectadas* y *Fácil Terminación por Equivalencia*, consulte [18].

CloSpan en vez de minar el conjunto de FCS directamente, primero genera todas las bases de datos proyectada del conjunto cerrado (LS), y entonces aplica la eliminación de las secuencias no cerradas, para generar exactamente el conjunto de FCS.

Basado en el concepto de *Fácil terminación por Equivalencia* son desarrollados dos métodos de poda del espacio de búsqueda: *Método del sub-patrón de retroceso y Método del súper-patrón de retroceso* (Figura 7).

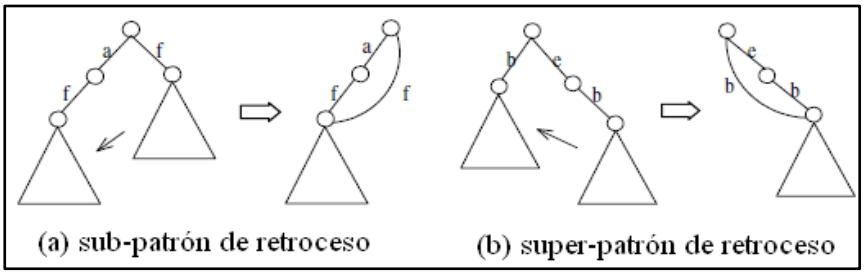

Fig. 7. Sub-patrón y super-patrón de retroceso

Estos métodos, se centran en la combinación de subárboles (ramas descendientes) en uno, bajo la condición de fácil terminación, y de esta manera evitan la poda redundante del espacio de búsqueda. Para el desarrollo de estos métodos se necesita mantener el conjunto de secuencias frecuentes cerradas, ya minadas, en la memoria ya que al utilizar el *Método del sub-patrón de retroceso* se chequea si un una secuencia recientemente encontrada pude ser absorbida por una secuencia frecuente cerrada ya minada, y/o al utilizar el *Método del súper-patrón de retroceso*, se chequea si una secuencia recientemente encontrada, puede absorber alguna secuencia frecuente cerrada candidata. Esto representa un gran costo en cuanto al consumo de memoria además que el espacio de búsqueda será muy grande, cuando exista un gran número de secuencias frecuentes cerradas.

Tras la mencionada combinación de subárboles, el Árbol de secuencias lexicográficas, anteriormente dicho, pude ser sustituido por un Retículo de Secuencias de prefijos como se muestra en la Figura 8, que además de ser un espacio de búsqueda, es una estructura de datos interna que almacena el conjunto LS. El resultado de CloSpan en el primer nivel es precisamente un retículo de secuencias de prefijos.

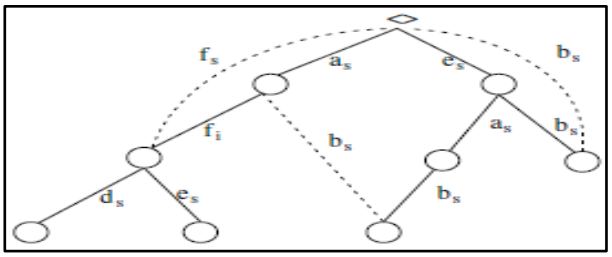

Fig. 8. Parte de un Retículo de Secuencias de Prefijos

Algoritmo CloSpan

El CloSpan puede ser descrito en dos pasos importantes: (1) genera el conjunto LS, y almacena estas en un retículo de secuencias de prefijos; y (2) se realiza una poda del espacio de búsqueda, para eliminar las secuencias no cerradas.

En el método 1 del Algoritmo 5 se presenta un paso de preprocesamiento necesario. Primero ordena todos los itemsets y borra los items no frecuentes y las secuencias vacías. Luego llama recursivamente al algoritmo CloSpan, para realizar una búsqueda en profundidad sobre el árbol de secuencias lexicográficas y construir el correspondiente retículo de secuencias de prefijos.

Algoritmo 5: CloSpan

Input: Base de datos D_s y *minsup.*
Output: Conjunto completo de secuencias cerradas *L*.
Método1:

1 Borra los items no frecuentes y las secuencias vacías, y ordena cada itemset de una secuencia en D_S;
2 $S^1 \leftarrow$ todas las 1-secuencias frecuentes;
3 $S \leftarrow S^1$;
4 **for each** secuencia $s \in S^1$ **do**
5 *CloSpan* (*s, D_s, minsup, L*);
6 eliminar las secuencias no cerradas de *L*

Input: Secuencia *s*, base de datos proyectada D_s y *minsup*.
Output: Retículo de búsqueda de prefijo *L*

Método2:

1 Comprobar si una secuencia encontrada s' de tal manera que $s \subseteq s'$ o $s' \subseteq s$, y $I(D_s) = I(D_{s'})$;
2 **if** existe un súper patrón o sub patrón **then**
3 modifica el enlace en *L,* **return***;*
4 **else** *insertar s en L*;
5 Explorar D_s y encontrar todos los items frecuentes α de tal manera que
(a) s puede ser extendida a ($s \Diamond_i \alpha$), o
(b) s puede ser extendida a ($s \Diamond_s \alpha$);
6 **if** no hay α disponible **then**
7 **return;**
8 **for each** α válido **do**
9 **call** *CloSpan*($s \Diamond_i \alpha$, *D, minsup, L*);
10 **for each** α válido **do**
11 **call** *CloSpan*($s \Diamond_s \alpha$, *D, minsup, L*);
12 **return**;

En el método 2 del Algoritmo 5, CloSpan es similar a PrefixSpan, aunque introduce una mejora importante, al utilizar las técnicas para la poda del espacio de búsqueda, mencionadas anteriormente. No obstante, aún resta un problema: como realizar las líneas de 1- 4 eficientemente. Chequear si todas las secuencias son sub-secuencias o súper-secuencias de una secuencia actual, se hace costoso, ya que el espacio de prueba es bastante grande. Entonces CloSpan utiliza el Algoritmo 6, para solucionar este problema, basado en tablas hash que contiene los pares ⟨I(Ds), s⟩, Figura 9. Sólo las secuencias cuyo tamaño de la base de datos proyectada sea igual al de la secuencia actual son chequeadas. En la línea 7 se

descubre la condición del súper-patrón de retroceso (s ⊆ s´). Este elimina todos los pares ⟨I(Ds), s⟩, que satisfacen la condición en la tabla hash y combina

sus correspondientes subárboles descendientes en *L* (retículo de secuencias de prefijos), esto significa borrar los subárboles duplicados.

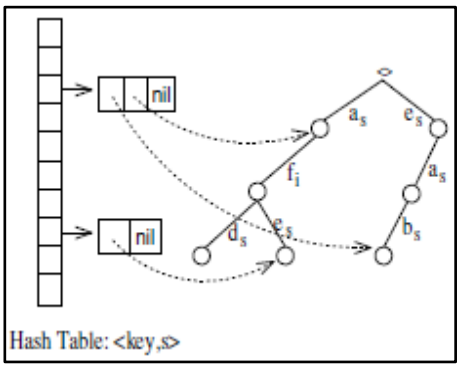

Fig. 9. Tabla Hash

Para eliminar las secuencias no cerradas del retículo de secuencias de prefijos, se debe chequear para cada secuencia *s,* si existe una súper secuencia *s´* de manera que *soporte(s)=soporte (s´).* Para esto, CloSpan adopta la misma estructura de datos de la Figura 9 y utiliza como función hash, la correspondiente suma de los identificadores de secuencias ($T(D_s)=T(s´)$), en vez de utilizar el soporte, ya que este puede ser el mismo para muchas secuencias no relacionadas. Sin embargo, como la equivalencia de $T(D_s)$ no implica equivalencia de soporte, para las secuencias que tienen el mismo valor de $T(D_s)$, sus soportes tienen que ser chequeados, para eliminar candidatos no válidos. Finalmente la contención es comprobada para ver si una secuencia puede ser absorbida. Esta forma de eliminación es similar al diseño para *Fácil Terminación por Equivalencia* con la ventaja de que la función hash es más fácil de calcular.

Algoritmo 6:

Input: Secuencia *s*, su clave *k*, y una tabla hash *H*.
Output: Una tabla hash actualizada *H.*

Método :

0 $l_{sup} \leftarrow 0$, $l_{sub} \leftarrow 0$;
1 indizar la tabla hash con la clave *k*;
2 encontrar una lista de pares *(k, s´);*
3 **for each** par*(k, s´)***do**
4 **if** *soporte(s)=soporte (s´)* **then**
5 **if** $s´ \subseteq s$ **then** $l_{sup} \leftarrow l_{sup} \cup \{(k, s´)\}$;
6 **if** $s´ \subseteq s$ **then** $l_{sub} \leftarrow l_{sub} \cup \{(k, s´)\}$;
7 **if** l_{sup} no está vacía **then**
borrar todos los pares en l_{sup} de *H*
combinar subárboles descendientes (de *s´* en l_{sup}) en L^1;
8 **if** l_{sub} no está vacía **then**
borrar todos los pares en l_{sub} de *H*
combinar subárboles descendientes (de *s´* en l_{sub}) en *L;*
9 insertar (k, s) en *H*;

Ventajas del CloSpan

En principio CloSpan no es comparable directamente con los algoritmos de minado de secuencias frecuentes presentados en el capítulo anterior, ya que CloSpan mina secuencias

frecuentes cerradas, considerando que los algoritmos antes expuestos minan secuencias no cerradas. Independientemente CloSpan supera a estos algoritmos, al adoptar nuevas técnicas de poda sobre espacio de búsqueda y proporcionar de esta manera una nueva visión para el minado escalable de grandes secuencias. Ya que estas técnicas de poda no modifican el algoritmo original, definiendo solamente la condición de *Fácil Terminación* sobre los subárboles, estas técnicas pueden ser extendidas a otros algoritmos para el minado de secuencias frecuentes ya conocidos. Sin embargo, debido a que CloSpan necesita mantener el conjunto histórico de secuencias candidatas cerradas, el consumo de memoria se hace muy alto, además de conducir a un enorme espacio de búsqueda, para el chequeo de secuencias cerradas cuando existen muchas secuencias frecuentes cerradas. Para solucionar este problema Wang y Han proponen un nuevo algoritmo para el minado de secuencias cerradas llamado *Bide*, sin la necesidad de mantener estos candidatos en memoria.

4.2 Bide: Eficiente algoritmo para minar secuencias frecuentes cerradas

El algoritmo Bide (BI-Directional Extension) propuesto en [19], es un eficiente algoritmo para el minado de secuencias cerradas sin la necesidad de mantener candidatos. Este adopta un novedoso esquema de control de cierre de secuencias llamado Extensión Bidireccional, y poda el espacio de búsqueda con mayor profundidad que los algoritmos presentados anteriormente, usando el método de poda BackScan y la técnica de optimización Scan-Skip. La extensión direccional de adelanto es usada para el crecimiento de patrones de prefijo y además para la comprobación de cierre de estos patrones; mientras que la extensión direccional de retroceso es usada tanto para la comprobación de cierre de los patrones de prefijo como para podar el espacio de búsqueda.

Algoritmo Bide

En el Algoritmo 8 se muestra el pseudocódigo para la implementación del Bide. Para minar eficientemente una base de datos de secuencias, el algoritmo Bide comienza con una exploración sobre la *DB*, para identificar las 1-secuencias frecuentes. Entonces una segunda búsqueda, construye la base de datos proyectada para estas secuencias. Sea *i* una secuencia, la proyección *i* de la DB es denotada como *P(i, DB),*que es el conjunto de subsecuencias que se compone de las secuencias en DB que contienen *i,* después de borrar los itemsets que aparecen antes de la primera ocurrencia de *i* en cada secuencia. Posteriormente, Bide busca cada base de datos proyectada y enumera las secuencias frecuentes, siguiendo una estrategia de crecimiento de patrones. Después de obtener las secuencias frecuentes, Bide aplica el *esquema de control de cierre,* llamado *Extensión Bidireccional,* para chequear si una secuencia frecuente es cerrada.

Algoritmo 7: Bide

Input: Una base de datos de secuencia *SDB*, Soporte mínimo *minsup.*
Output: conjunto completo de secuencias frecuentes cerradas FCS.

```
1  FCS = 0;
2  F1 = 1-secuencias frecuentes;
3  for each (1-secuencia f1 en F1) do
4  SDB|f1 = Base de datos pseudo proyectada;
5  for each (f1enF1) do
6  if (!BackScan (f1, SDB|f1));
7  BEI = items con extensión hacia atrás;
8  call bide(SDB|f1, f1, minsup, BEI, FCS);
9  return FCS;
```

// Parámetros: *Sp_SDB*:Base de datos de secuencias proyectadas, *Sp*: secuencia de

prefijo
//*minsup*: soporte mínimo, *BEI:* números de items con extensión hacia atrás.
bide(*Sp_SDB, Sp, minsup, BEI, FCS*)
Output: Conjunto actual de secuencias frecuentes *FCS.*

```
1   LFI= items localmente frecuentes;
2   FEI = |{z en LFI , tal que: z.sop = sop^SDB(Sp)}|;
3   if ((BEI + FEI)== 0 )
4   FCS = FCS ∪ {Sp};
5   for each (i en LFI) do
6   Sp^i = <Sp, i>
7   SDB^Spi = Base de datos pseudo proyectada;
8   for each (i en LFI) do
9   if (!BackScan (Sp^i, SDB^Spi ))
10  BEI = items con extensión hacia atrás;
11  call bide(SDB^Spi, Sp^i, minsup, BEI, FCS);
```

Enumeración de secuencias frecuentes

Asumiendo que existe un orden lexicográfico, conceptualmente el espacio de búsqueda para el minado de secuencias constituye un árbol de secuencia, donde al eliminar las secuencias no frecuentes, los nodos restantes en el retículo, forman un árbol de secuencias frecuentes lexicográficas. Bide atraviesa el árbol de secuencia, en una estricta búsqueda en profundidad, para encontrar las secuencias frecuentes.

Los nodos en un árbol de secuencias pueden ser tratados como secuencias de prefijos, a partir los cuales son generados los conjuntos descendientes, por la adición de items localmente frecuentes. Para calcular los item localmente frecuentes con respecto a cierto prefijo, Bide usa el método de la Pseudo- Proyección, mencionado en el capítulo 2.

El orden de la enumeración de las secuencias frecuentes está en correspondencia con la búsqueda en profundidad del árbol de secuencias.

Extensión bidireccional

El algoritmo CloSpan presentado anteriormente, necesita mantener el conjunto de secuencias frecuentes cerradas, ya minadas, en memoria. Como se puedo ver, estos *esquemas de control de cierre* consumen mucha memoria y el espacio de búsqueda será muy grande, cuando exista un gran número de secuencias frecuentes cerradas. Para evitar esto, Bide introduce un nuevo esquema de control de cierre de secuencias: *Extensión Bidireccional.*

De acuerdo a la definición de secuencia frecuente cerrada si una *n*-secuencia, $S=e_1e_{2...}e_n$, es no cerrada, debe existir al menos un item, *e´,* el cual puede ser usado para extender la secuencia *S* a una nueva secuencia, *S´.* Durante esta extensión si el itemset *e´* ocurre antes que e_n, *e* se llamará *itemset de extensión hacia adelante,* y *S´* una *secuencia de extensión hacia atrás* con respecto a *S.* Mientras que si el itemset *e* ocurre después que e_n, e se llamará *itemset de extensión hacia atrás* y *S´* una *secuencia de extensión hacia atrás* con respecto a *S.* Luego el siguiente teorema será clave para el control de cierre de una secuencia.

Teorema1 (*control de cierre de Extensión Bidireccional*): Si no existen *itemset de extensión hacia adelante o hacia atrás* con respecto a una secuencia de prefijo S_p, entonces S_p debe ser una secuencia cerrada, de lo contrario S_p es no cerrada.

Por el Teorema 1, sabemos que para juzgar si una secuencia de prefijo frecuente es cerrada necesitamos chequear si existe algún *itemset de extensión hacia adelante o hacia atrás.* En [19] los autores presentan varias definiciones para realizar esta comprobación de una forma relativamente fácil.

5 Computación paralela

La computación paralela es la solución más factible para que un computador adquiera la capacidad de procesar las aplicaciones más sofisticadas en un espacio de tiempo razonable. Para evaluar entonces el rendimiento de un sistema paralelo abordaremos algunos indicadores como la eficiencia, el aumento de velocidad o Speedup y la escalabilidad entre otros. También abordamos otras cuestiones que deben ser bien administradas para maximizar estos indicadores, como los gastos u overhead de comunicación y el equilibrio de carga.

5.1 Speedup, escalabilidad y balance de carga

El objetivo fundamental de la computación paralela es lograr reducir el tiempo de ejecución de una aplicación, en un factor que sea inversamente proporcional al número de procesadores usados [22]. Entonces si usamos dos procesadores, el tiempo de ejecución se debe reducir a la mitad con respecto al tiempo de ejecución que requiere un procesador. Cualquier algoritmo que alcance este objetivo se dice que es escalable. En otras palabras se puede decir que la escalabilidad se evalúa con relación al incremento del tamaño tanto del sistema paralelo (# de CPU, cantidad de memoria, etc.), como del problema. Sean Time1 el tiempo para resolver un problema de tamaño P con un sistema paralelo que tiene P procesadores y Time2, el tiempo para resolver un problema de tamaño Q con un sistema paralelo de Q procesadores, tal que P> Q, entonces la escalabilidad se mide como Sc = Time1/ Time2. En condiciones ideales Sc = 1, lo cual significa que la escalabilidad es lineal, es decir, el aumento tanto del tamaño del sistema como del problema, podrían mantener el tiempo de procesamiento constante [26].

Para concretar este objetivo, aparece también el término de aumento de velocidad o Speedup, que se define como la razón del tiempo de ejecución de un único procesado y el tiempo de ejecución de n procesadores configurados en paralelos, como muestra la siguiente expresión:

$$Speedup(n) = T(1)/T(n) \tag{4}$$

Hay ocasiones en que la escalabilidad de una aplicación no se alcanza de una manera sencilla, debido a la existencia de tres factores principalmente. En primer lugar la aplicación puede tener una región que se ejecute secuencialmente. Sea Ts el tiempo requerido por esta región y Tp el tiempo requerido por una región paralela, el Speedup viene dado por:

$$Speedup(n) = \frac{Ts + Tp}{Ts + \frac{Tp}{n}} \leq \frac{T(1)}{Ts} \tag{5}$$

Lo que quiere decir que el *Speedup* está limitado por la razón del tiempo de ejecución que requiere una región secuencial. Esto se conoce como la Ley de Amdahl's. En segundo lugar está el alto grado de comunicación o coordinación que se produce entre los procesadores. Y el tercer impedimento que afecta la escalabilidad es el pobre *balance de carga*. Por ejemplo si un procesador tiene la mitad de la carga del trabajo paralelo, el Speedup se limitará aún factor de 2, sin importar cuantos procesadores haya.

Para alcanzar un buen *balance de carga* se deben dividir las tareas en bloques de igual tamaño, de forma tal que cada procesador tenga la misma cantidad de trabajo. Sin embargo, lograr esto se hace realmente difícil cuando se trabaja sobre una máquina de memoria distribuida o cuando se desconoce la carga de trabajo hasta el momento de ejecución. Entonces se requiere de un esquema de balance de carga dinámica, donde el trabajo asignado a cada procesador se realiza en tiempo de ejecución.

5.2 Eficiencia

Como pudimos ver anteriormente el Speedup mide solamente la efectividad en la explotación del paralelismo, sin medir la eficiencia del proceso paralelo [5]. La eficiencia de un proceso paralelo (Ef) puede ser determinada por (6), por ejemplo: si un sistema paralelo con 4 procesadores alcanza un Speedup de 3, se puede decir que el proceso paralelo tiene una eficiencia de 75%. Por lo tanto la Ef es un indicador extremadamente importante para evaluar la escalabilidad de un proceso paralelo determinado, con respecto al tamaño del problema en cuestión.

$$Ef = Speedup / p \tag{6}$$

5.3 Memoria compartida y memoria distribuida

La mayoría de las máquinas paralelas actuales, se dividen en dos categorías básicas [22]:

1. *Máquinas de memoria compartida:* tienen un único espacio de memoria compartido, al cual puede acceder cualquier procesador, Figura 10.
2. *Máquinas de memoria distribuida:* el sistema de memoria es empaquetado con nodos individuales de uno o más procesadores, y se necesita comunicación para un procesador pueda acceder a la memoria de otro, Figura 11.

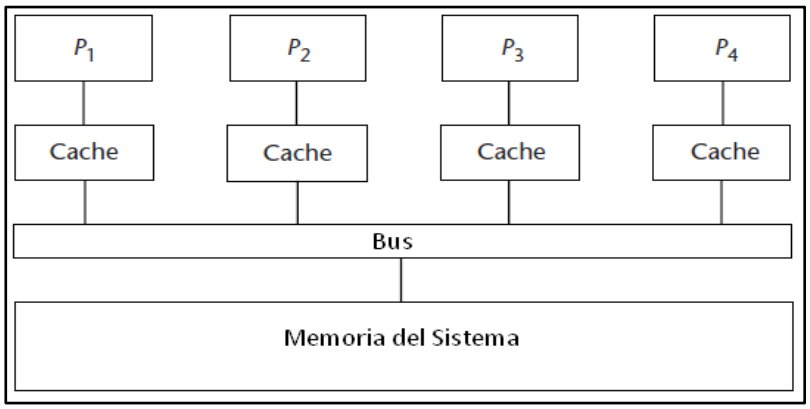

Fig. 10. Memoria compartida

En las máquinas de memoria compartida, cada procesador tiene asignada una cache privada, interconectada a una memoria compartida global, a través de un bus, como se muestra en la Figura 10. Esta organización es típicamente llamada Multiprocesador Simétrico (SMP por sus siglas en inglés). La sincronización de acceso a la estructura de datos compartidos es un tema importante en los sistemas de memoria compartida. Le corresponde al programador asegurar que las operaciones realizadas por diferentes procesadores sobre la estructura de datos compartidos, dejen a esta en estado coherente. Este tipo de máquina no es escalable ante un gran número de procesadores, debido a contención del Bus.

Esta limitación en cuanto a la escalabilidad en las memorias compartidas ha llevado a los diseñadores a utilizar memorias distribuidas [22]. En las memorias distribuidas la memoria global compartida es sustituida por pequeñas memorias locales adjuntas a cada procesador (Figura 11), donde la comunicación entre las memorias de los procesadores es a través de una red de comunicación. La ventaja de esta arquitectura es que el acceso a los datos locales es mucho más rápido. En este tipo de arquitectura, el intercambio de datos entre

procesadores se realiza mediante pases de mensajes, lo cual constituye un problema en cuanto al gasto (*overhead*) de comunicación entre los procesadores.

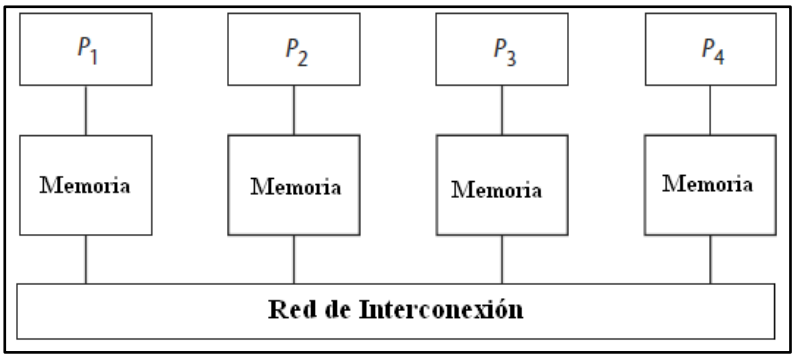

Fig. 11. Memoria distribuida

5.4 Gasto (overhead) de comunicaciones

El gasto u overhead de comunicaciones entre procesadores depende en gran medida de la arquitectura de la máquina. Para explicarlo con más claridad nos basaremos en la arquitectura de memoria distribuida, descrita anteriormente.

En una máquina de memoria distribuida, el tiempo de transmisión de los mensajes entre procesadores está en dependencia de la topología de la red de interconexión. En la Figura 12 se muestran dos topologías de red diferentes: topología de Anillo y topología de hipercubo [5].

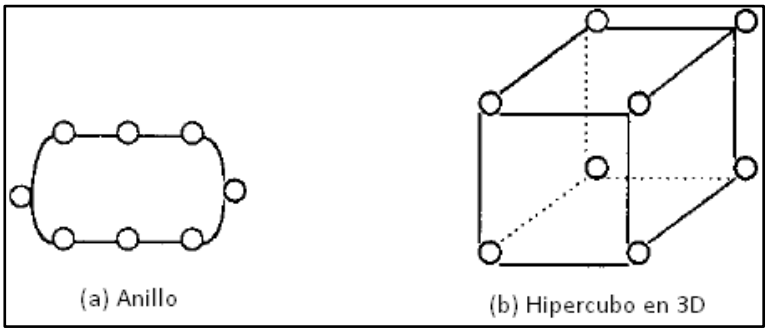

Fig. 12. Topología de red de interconexión

5.5 Paralelismo espacial y temporal

El llamado paralelismo temporal o pipelining se refiere a la ejecución de tareas como una cascada de subtareas, donde cada subtarea es ejecutada por una unidad funcional (procesadores); estas unidades trabajan al mismo tiempo de manera superpuesta. Los datos procesados por un procesador Pi, son enviados al próximo procesador Pi+1 y Pitoma un nuevo dato para procesarlo, similar al proceso de ensamblado de un automóvil. Cada procesador puede ser visto "un especialista", en el sentido que siempre realiza la misma sub-tarea [5]. En este sentido no se necesita sincronización durante la ejecución paralela. Idealmente siempre se debe tratar de utilizar el paralelismo asincrónico porque esto nos da más oportunidades para la escalabilidad.

Por otra parte el paralelismo espacial se refiere a la ejecución simultánea de tareas por varios procesadores. En un momento dado estos procesadores pueden ejecutar la misma tarea (o instrucción) o tareas independientes. Con el fin de lograr un tiempo de ejecución

pequeño, los gastos de ejecución de estas tareas en paralelo deben reducirse grandemente. Entonces para este tipo de paralelismo si es fundamental lograr un buen sincronismo y un buen balanceo de carga [24].

5.6 Paralelismo de tareas y paralelismo de datos

Las dos formas principales de aplicar el paralelismo en los algoritmos de Minería de Datos son: el Paralelismo de Tareas o de Control y el Paralelismo de Datos o SPMD (Simple Program Multiple Data).

De las dos formas, la de menor complejidad de implementar es el paralelismo de Datos, debido a las facilidades de uso y posibilidades de paralelización automática, brinda mejor escalabilidad ante grandes volúmenes de datos, a costa de incrementar CPU + Memoria en correspondencia con el aumento de los datos, aunque esto tiene sus limitaciones, tanto financieras como prácticas, además presenta mayor independencia respecto a la arquitectura de hardware [26].

5.7 Partición de datos

La partición de datos es la base para aplicar eficientemente el paralelismo de datos, donde un conjunto de datos es particionado en varios subconjuntos de datos, que son asignados a distintos procesadores, de forma tal que cada procesador pueda realizar la misma operación solamente sobre su subconjunto local.

Con el fin de distribuir un conjunto de datos entre P procesadores disponibles, los datos pueden ser particionados horizontal o verticalmente (Ver Figura 13). En el particionado horizontal el conjunto de tuplas es particionado en P subconjuntos de tuplas, donde cada subconjunto contiene todos los atributos para las correspondiente tuplas. Para el método del particionado vertical, el conjunto de atributos de la relación es particionado en P subconjuntos de atributos, donde cada uno contiene los correspondientes valores de atributos para todas las tuplas de la relación [5].

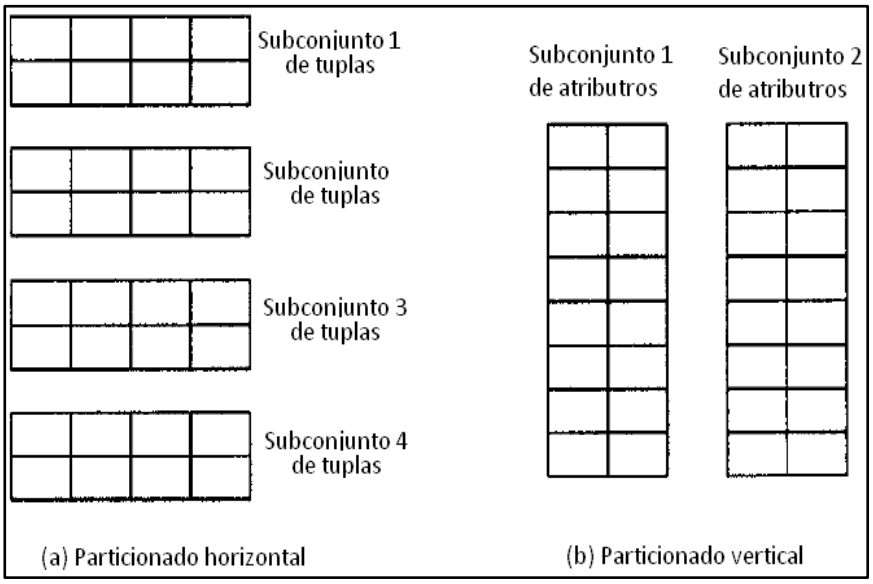

Fig. 13. Tipos de particionado

6 Algoritmos paralelos para el minado de secuencias frecuentes

El enorme Espacio de Búsqueda hace que la minería de secuencias frecuentes sobre grandes bases de datos sea un problema de alto nivel computacional. El desarrollo de algoritmos series de minado de secuencias frecuentes (MSF), está limitado por la capacidad de cálculo de un solo procesador y el espacio finito de memoria; por lo tanto ellos no son lo suficientemente escalables ante enormes bases de datos. Para lograr que un algoritmo de Minería de Datos sea escalable ante grandes volúmenes de datos, muchas aplicaciones y sistemas utilizan la computación paralela (high performance computing) [16], que alivian el cuello de botella de los actuales algoritmos de MSF. Cada vez más la computación paralela se está viendo como el único método eficaz para la rápida solución de grandes problemas de cómputo. La aparición de ordenadores paralelos como multiprocesadores de escritorio y clusters de estaciones de trabajo han hecho posible la aplicación de algoritmos paralelos. Esto también prepara el escenario para un crecimiento sustancial de la programación paralela.

Si bien el minado en paralelo de reglas de asociaciones ha atraído una gran atención, se ha trabajado menos en la minería en paralelo de patrones frecuentes. En este sentido, Shintani y Kitsuregawa [7] proponen tres algoritmos paralelos para computadores de memoria distribuida basados en el algoritmo serie GSP: el NPSPM, SPSPM y el HPSPM. Los tres Algoritmos particionan la base de datos entre todos los nodos de procesadores. Zaki en [14] propone el pSPADE, basado en el algoritmo serie SPADE. El pSPADE fue implementado sobre una máquina SGI Origin 2000, que presenta una arquitectura de memoria compartida; además trata al espacio de patrones como un árbol de clases independientes basadas en sufijos. Cong y Han en [21] proponen el algoritmo Par-CSP basado en el algoritmo de minado de *FCS:*BIDE, y que corre sobre un sistema de memoria distribuida. Par-CSP hace uso de la técnica de búsqueda en profundidad y del enfoque de divide y vencerás del algoritmo BIDE. Una taxonomía donde se resumen los principales algoritmos paralelos en la minería de secuencias se muestra en la Figura 14.

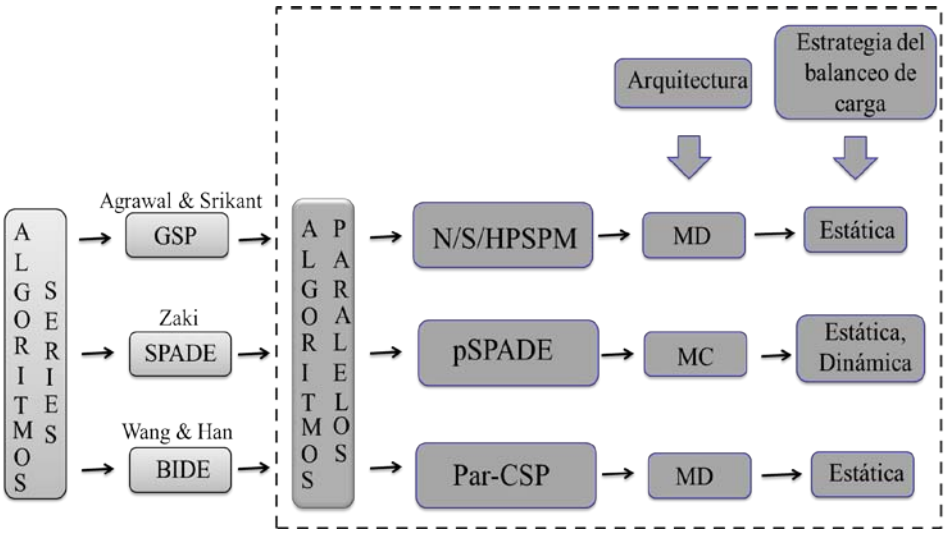

Fig. 14. Taxonomía de los algoritmos paralelos típicos en la minería de secuencias

6.1 Algoritmos basados en GSP

Basados en el algoritmo serie GSP, para el MSF, Shintani y Kitsuregawa en [7], proponen tres algoritmos paralelos para este tipo de minería, sobre un ambiente nada compartido: el NPSPM (Non Partitioned Sequential Pattern Mining), el SPSPM (Simply Partitioned Sequential Pattern Mining) y el HPSPM (Hash Partitioned Sequential Pattern Mining). En NPSPM, las secuencias candidatas son sencillamente copiadas entre todos los nodos y cada

nodo determina las secuencias frecuentes comparando los valores del soporte entre todos los otros nodos. Aunque cada nodo puede trabajar independientemente durante el proceso de conteo del soporte, cada nodo tiene que examinar todas las secuencias candidatas. Ya que NPSPM copia el conjunto completo de datos sobre cada nodo, esto puede provocar problemas de desbordamiento de memoria ante grandes bases de datos. Los dos algoritmos restantes particionan las secuencias candidatas sobre los nodos, lo cual puede explotar de manera eficiente la memoria total del sistema. SPSPM aunque utiliza toda la memoria del sistema, provoca excesivas comunicaciones, ya que cada procesador simple realiza un broadcast a todos los demás procesadores, para obtener el soporte global. HPSPM utiliza una estrategia más inteligente para particionar las secuencias candidatas usando un mecanismo hashing. Este además reduce la cantidad de comunicaciones para contar el soporte global. Estos algoritmos fueron desarrollados sobre una máquina IBM SP2 con 16 nodos y emplea una arquitectura de memoria distribuida.

Como estos tres algoritmos son muy parecidos, salvo por las diferencias anteriores, en lo siguiente presentaremos el HPSPM, que es el de mayor rendimiento entre los tres.

6.1.1 HPSPM

HPSPM particiona las secuencias candidatas entre los nodos usando una función hash, lo cual elimina la necesidad del envío de secuencias de datos y puede reducir la carga de trabajo de comparación. El Algoritmo 8, muestra el pseudocódigo para el HPSPM, usando la notación de la Tabla 4.

Cada nodo trabaja de la siguiente manera:

1. Genera las secuencias candidatas:
   - Cada nodo genera C_K a partir de L_{K-1}.
   - Aplica la función hash a los candidatos en C_K y determina el ID del nodo destino. Si el ID es propio, inserta los candidatos en la tabla hash(C_K^P).
2. Explora la base de datos de secuencias y cuenta el valor del soporte:
   – Cada nodo lee las secuencias de la base de datos, desde su disco local y genera una nueva secuencia *s´,* mediante la selección de todos los items que están contenidos en C_K.
   - Genera *k*-secuencias a partir de *s´* y aplica la misma función hash usada en la fase 1. Deduce el nodo destino ID y envía la *k*-secuencia a este.
   - Para las secuencias recibidas de otros nodos y las generadas localmente, cuyo ID es igual al propio ID del nodo, busca en la tabla hash de candidatos. Si acierta, el incrementa el valor de soporte.
3. Determina las secuencias largas:
   - Después de leer todas las secuencias, cada nodo puede determinar individualmente si cada secuencia candidata en C_K^P, satisface la condición de soporte mínimo.
   - Cada nodo envía L_K^P a un nodo central, donde todas las *k*-secuencias largas son derivadas.
4. Si L_K está vacía, el algoritmo termina. De lo contrario. El nodo coordinador envía L_K a todos los nodos.

Tabla 4. Notación para el algoritmo HPSPM

L_K	Conjunto de todas las *k*-secuencias largas
C_K	Conjunto de todas las *k*-secuencias candidatas
D^P	Secuencias de clientes almacenadas en el disco local del nodo *p*
C_K^P	Conjunto de todas las *k*-secuencias asignadas al nodo *p*
L_K^P	Conjunto de *k*-secuencias largas derivadas de C_K^P

Algoritmo 8: HPSPM

1 **for** $k \geq 2$
2 C_K = las candidatas de tamaño *k* generadas a partir de L_{K-1};
3 { C_K^P }= todos los *k*-itemsets candidatos, cuyo valor hash corresponde al nodo *p-th*;
4 **forall** ($s \in D^P$) **do**
5 s' = Seleccionar todos los items contenidos en C_K de la secuencias de cliente
6 **forall** (k-secuencia x $\in$ s´) **do**
7 Determinar el identificador del nodo destino mediante la aplicación de la misma función hash que se utiliza en la partición de secuencias candidatas y enviar la *k*-secuencia a este;
if esta secuencia ya está en este ID **then**
Incrementar el contador del soporte para esta secuencia;
Recibir *k*-secuencias de otros nodos e incrementar el contador del soporte para esta secuencia;
8 **end**;
9 **end**;
10 { L_K^P } = Todas las candidatas en C_K^P con soporte mínimo;
11 Enviar L_K^P al nodo coordinador;
// El nodo coordinador está formado por $L_K = \bigcup_P L_K^P$ y las envía a todos los nodos.
12 Recibir L_K del nodo coordinador;

Desventajas de estos algoritmos

La principal limitación de estos algoritmos paralelos es que realizan repetidos pases sobre las particiones que se encuentran en el disco, incurriendo en altos gastos de I/O. Además los esquemas implican intercambios entre las particiones de la base de datos, durante cada iteración, provocando gran cantidad de comunicaciones y operaciones de sincronización. Ellos también usan complicadas estructuras hash, lo que implica un gasto adicional en la búsqueda.

6.2 Algoritmo pSPADE

Zaki en [14], basado en el SPADE [11], propone el pSPADE algoritmo paralelo para el descubrimiento de secuencias frecuentes, implementado sobre un sistema de memoria compartida. El pSPADE descompone el espacio de búsqueda inicial en pequeñas clases basadas en sufijos. Cada clase puede ser generada en la memoria principal mediante técnicas de búsqueda eficiente y simples operaciones de unión. Además cada una de ellas puede ser procesada de forma independiente en cada procesador sin requerir sincronización. Sin embargo, el balance de carga dinámica entre clases y dentro de la clase debe ser explotado

para asegurar que cada procesador reciba la misma cantidad de trabajo. El algoritmo fue implementado sobre 12 procesadores SGI Origin 2000 con sistema de memoria compartida.

PSPADE: Diseño e implementación

El algoritmo pSPADE se entiende mejor cuando nos imaginamos el proceso como la expansión dinámica e irregular de un árbol de clases basadas en sufijo. Estas clases padres sólo son visibles al comienzo del proceso. Como el algoritmo se está implementando sobre una máquina de memoria compartida, solamente hay una copia sobre el disco de la base de datos con formato vertical. Se podrá acceder a esta desde cualquier procesador, a través de un descriptor local. Teniendo en cuenta que cada clase en el árbol puede ser desarrollada independientemente, la cuestión fundamental es lograr un buen balance de carga, de modo que cada procesador reciba la misma cantidad de trabajo.

Para implementar este algoritmo el autor compara dos técnicas, presentadas en el capítulo anterior, el paralelismo de datos y paralelismo de tareas. En el paralelismo de datos los procesadores trabajan sobre distintas particiones de la base de datos, pero procesa sincrónicamente el árbol de clases global. Este esquema es basado en el paralelismo intra-Clases (dentro de las clases). Mientras que con el paralelismo de tareas los procesadores comparten la base de datos, pero trabajan sobre diferentes clases en paralelo, procesando asincrónicamente el árbol de clases. Este esquema es basado en el paralelismo inter-Clases.

Aplicando paralelismo de datos

En el paralelismo de datos aplicado a la implementación de este algoritmo, a cada nodo le corresponde una clase equivalente de secuencias frecuentes, la cual necesita ser expandida al próximo nivel. Este tipo de paralelismo se presenta de dos maneras, ya que se puede particionar la *id-list* horizontal o verticalmente. Cuando se particiona horizontalmente se divide cada id-list en bloques horizontales que se asignan a cada procesador (*paralelismo de id-list*). Mientras que en la partición vertical se asignan por separado las *id-lists* a cada procesador (*paralelismo de unión*). Ninguno de los dos modos es óptimo por el excesivo overhead de sincronización que requieren. Para más detalle consulte [14].

Aplicando paralelismo de tareas

En el Paralelismo de tareas todos los procesadores tienen acceso a una copia de la base de datos, pero ellos trabajan sobre clases separadas. Es importante notar que como se realiza una búsqueda en profundidad para enumerar las secuencias frecuentes, en cada clase padre, es necesario organizar cada una de estas independientemente, para un mejor balanceo de carga. En este sentido los autores presentan dos tipos de balanceo de carga 1) Balanceo de carga estática y 2) Balanceo de carga dinámica.

Balanceo de carga estático (SLB por sus siglas en inglés*)*: Dado un conjunto de clases padres $C = \{C_1, C_2, C_3\}$ y dos procesadores P_0 y P_1. Se necesita repartir estas clases entre todos los procesadores de manera que se minimice el desbalanceo de carga. El balanceo de carga es alcanzado, asignando una función de peso a la clase C_i, basado en el número de elementos de la clase. Cada clase es organizada a partir de su peso (en orden decreciente), y se asigna cada clase en turno al procesador de menor carga. Los empates se rompen seleccionando al procesador de menor identificador. Estos pasos son realizados al mismo tiempo sobre todos los procesadores, ya que todos ellos tienen acceso a *C.*

Una vez repartidas las clases padres, el cálculo procede de una manera puramente asincrónica, ya que no es necesario ningún tipo de sincronización, o información compartida entre los procesadores. En el ejemplo presentado, se nota que la técnica de *SLB*, sufre de desbalanceo de carga, ya que después de procesar C_1, P_0 estará ocupado Procesando C_3, mientras que después de procesar C_2, P_1 no tendrá más carga de trabajo, debido a la naturaleza irregular del árbol de clases.

Algoritmo 9: Balanceo de carga dinámico entre clases

Input: Base de datos de secuencias *D* y Soporte mínimo *minsup*.

```
1  C= {clases padres C_i= [X_i]};
2  Sort –on-Weight(C);
3  shared int clase_id = 0;
4  for each (procesador P_j) do en paralelo
5     for (i =0; i< | C | ; i++)
6        if (compare_and_swap (classid, i, i+1))
7  Enumerate-Frequent-Seq(C_i);
```

Balanceo de carga dinámico entre clases (CDLB por sus siglas en inglés*):* Esta técnica ofrece un balanceo de carga más óptimo que el *SLB*, ya que cada procesador puede seleccionar dinámicamente una nueva clase, de la lista de clases padres que aún no han sido procesadas. Esta técnica también usa el peso de clases. Primero se organizan las clases padres en orden decreciente de sus pesos, formando una cola de tareas de lógica central de clases independientes. Ya que las clases son organizadas a partir de sus pesos, los procesadores primero trabajan sobre las clases más largas antes de procesar otras más pequeñas, lo cual ayuda a alcanzar un mayor grado de balanceo de carga. En el Algoritmo 9 se presenta el pseudocódigo para un *CDLB*. La función *compare-and-swap*, compara *classid* con *i.* Si ellos son iguales, este sustituye *classid* por *i+1*, y retorna un 1, sino retorna un 0.

Para el ejemplo presentado *CDLB* no ofrece ninguna mejora respecto *SLB*; sin embargo, para un gran número de clases padres si presenta una clara ventaja, ya que un procesador agarra una nueva clase solamente cuando este ha procesado su clase actual. De esta forma sólo los procesadores sin carga de trabajo, procesaran nuevas clases, mientras otros continúan procesando su clase actual, ofreciendo una buena utilización de los procesadores.

Balanceo de carga dinámico recursivo (RDLB por sus siglas en inglés*):* Aunque *CDLB* presenta mejoras ante *SLB*, sólo puede ser aplicado a nivel de inter-Clases, lo cual puede ser demasiado grueso para alcanzar un buen balanceo de carga de trabajo. *RDLB* soluciona este problema, explotando ambas métodos de paralelismo, tanto inter-Clases como intra-Clases.

Para ver como explotar el paralelismo intra-Clases, analizaremos el peor de los casos, cuando *P-1* procesador*es* están libres y solamente uno está ocupado, especialmente si la última clase tiene un profundo árbol de cálculo. Para superar esto, *RDLB* aporta un mecanismo para que los procesadores libres, se unan a otro que este aún ocupado. Como cada clase es independiente, se pueden tratar de la misma manera que son tratadas las clases padres, y entonces diferentes procesadores pueden trabajar sobre diferentes clases en el mismo nivel. El Algoritmo 10 muestra el pseudocódigo del pSPADE con el uso de un esquema *RDLB.*

Algoritmo 10: pSPADE

Input: Base de datos D y soporte mínimo minsup

```
1  sharedint FreeCnt = 0;  // Número de procesadores libres
2  sharedint GlobalFlg = 0;  // Hay más trabajo?
3  sharedint GlobalQ; // Lista global de clases
4  GlobalQ = C = {clases padres C_i = [X_i]};
5  Sort-on-Weight (C);
6  Process-GlobalQ();
7  FreeCnt ++;
8  while (FreeCnt ≠ P)
```

```
 9   if (GlobalFlg) then
10         FreeCnt - -; Process-GlobalQ(); FreeCnt ++;

Process- GlobalQ():
11  sharedint clase_id = 0;
12  parallel for (i = 0; i<GlobalQ.size(); i ++ )
13  if (compare_and_swap (classid, i, i+1))
14  RDLB-Enumerate-Frequent-Seq(C_i);
15  GlobalFlg = 0;

RDLB-Enumerate-Frequent-Seq(PrevL):
16  for (; PrevL ≠ 0; PrevL = PrevL.next())
17  if (FreeCnt> 0) then
18  Add-to-GlobalQ(PrevL.next()); GlobalFlg = 1;
19  NewL = NewL U Get-New-Classes(PrevL.item());
20  if (NewL ≠ 0) then
21  RDLB-Enumerate-Frequent-Seq(NewL);
```

En el Algoritmo 10, primeramente se insertan las clases padres en una lista de clases global, donde cada procesador selecciona una clase, hasta que todas hayan sido tomadas. La función *Process-GlobalQ ()* de la línea 6 es similar al lazo principal de *CDLB*. Cada clase padre es procesada mediante una búsqueda en profundidad. Si un procesador terminó de procesar su clase y no hay más clases padres a la izquierda, este incrementa la variable compartida *FreeCnt,* y espera por más trabajo. Cuando un procesador está procesando las clases en el mismo nivel, chequea periódicamente si hay algún procesador libre (línea 17). Si es así este mantiene una clase para sí mismo e inserta las restantes clases en el nivel (*PrevL*), en *GlobalQ.* Este procesador continúa trabajando en las clases generadas (*NewL*), hasta que se detecte el primer procesador libre. Todas las intersecciones de *id-list* son desarrolladas en la rutina *Get-New-Classes ().* Cuando un procesador en espera detecta que hay más trabajo (*GlobalFlg* = 1), este comienza a trabajar sobre clases en *GlobalQ.* Finalmente cuando no hay más trabajo en la cola global, *FreeCnt* se iguala al número de procesadores *P* y se detiene el cálculo.

Según el análisis realizado para este algoritmo en particular, se puede notar que cada procesador puede acceder directa e igualmente a la memoria común del sistema, lo que unido al hecho de que el ancho de banda es finito puede limitar la escalabilidad. También se concluyó que el paralelismo de datos, no presenta un buen rendimiento debido al excesivo gasto en cuanto a la sincronización, a diferencia del paralelismo de tareas que es totalmente asincrónico y donde se introducen eficientes técnicas para lograr un buen balanceo de carga de trabajo.

6.3 Par-CSP

Cong y Han en [21] proponen el algoritmo Par-CSP, que es el primer algoritmo dirigido al minado paralelo de secuencias frecuentes cerradas. Este algoritmo se basa en el algoritmo serie Bide, para el minado de FCS y está diseñado para la ejecución sobre un sistema de memoria distribuida. Par-CSP particiona el trabajo entre los procesadores, explotando la propiedad de divide y vencerás, que minimiza el overhead de las comunicaciones entre procesadores. Además aplica una programación dinámica para minimizar el tiempo que permanece desocupado un procesador y una técnica llamada muestreo selectivo para solucionar el problema del desbalanceo de carga. El muestreo selectivo predice con exactitud el tiempo relativo de las subtareas y de esta manera permite una distribución uniforme del trabajo entre los procesadores.

Descomposición y programación de tareas

La estrategia utilizada por los autores para el minado en paralelo de FCS es la siguiente:

1. Cada procesador cuenta la ocurrencia de 1-secuencias en una parte diferente de la base de datos. Una operación global de suma-reducción es ejecutada para obtener el soporte global. Entonces las 1-secuencias frecuentes son identificadas.
2. Para cada 1-secuencia frecuente, es construida una representación muy compacta de la proyección del conjunto de datos, llamada *pseudo proyección.* Esto se hace en paralelo mediante la asignación de diferentes partes del conjunto de datos, a cada procesador. Las pseudo proyecciones están comunicadas a cada procesador mediante un *broadcast* todos a todos.
3. Se realiza una programación dinámica para distribuir el procesamiento de las proyecciones entre los procesadores.

En la implementación de este algoritmo, los autores proponen realizar el *broadcast* usando una estructura de anillo virtual, que resulta más eficiente. En esta estructura un procesador *I* recibe información del procesador (*I-1*) y envía solo información al procesador (*I+1*). Entonces asumiendo un total de procesadores *N,* el *broadcast* de todos a todos, es realizado en (*N-1*) pasos de envío-recibo, lo cual en conjunto no consume más del 0.5%del tiempo de minado.

Para reducir el desbalance de carga, Par-CSP usa una programación dinámica. Para la implementación de este algoritmo en particular, hay un procesador máster que mantiene una cola de identificadores de pseudo proyecciones. A cada uno de los otros procesadores se le asigna inicialmente una proyección; entonces cada procesador completa la minería de la proyección y envía una petición al procesador máster para otra proyección, el cual responde con el índice de la próxima proyección y la elimina de la cola. Este proceso continuo hasta que la cola de proyecciones esté vacía. Las solicitudes y respuestas para y desde el procesador máster, son mensajes cortos y el tiempo de comunicación es usualmente insignificante al tiempo de minado.

La programación dinámica es bastante efectiva cuando las subtareas son de tamaños similares y numerosos. Sin embargo, en muchos de los casos, la programación dinámica no puede lograr el balance de carga.

Estimación relativa del tiempo de minería

El enfoque que utiliza Par-CSP para mejorar efectividad de la programación dinámica es identificar cuales proyecciones requieren un mayor tiempo de minado y entonces descomponer estas. Para ello se necesita estimar el tiempo relativo de minado de las proyecciones. La estrategia que proponen los autores es la de utilizar el muestreo del tiempo de ejecución.

La estrategia más natural de muestreo es el llamado *muestreo aleatorio,* mediante el cual se colecciona, al azar, un subconjunto de secuencias en la base de datos, se calculan sus proyecciones, y se usa el tiempo de minado de los subconjuntos para estimar el tiempo de minado de cada proyección. Sin embargo, es tipo de muestreo no resulta muy exacto.

Como alternativa se introduce una nueva estrategia de muestreo llamada *muestreo selectivo*, que ha demostrado ser bastante exacta en la identificación de proyecciones que requieren grandes tiempos de minado. En lugar de seleccionar aleatoriamente un subconjunto de secuencias, el *muestreo selectivo* usa componentes de cada secuencia del conjunto de datos. El *muestreo selectivo* primero descarta las 1-secuencias no frecuentes y luego descarta las últimas *l* 1-secuencias frecuentes de cada secuencia. El número *l* es calculado mediante la multiplicación de una fracción *t* dada, y la longitud media de las secuencias en el conjunto de datos [21].

Algoritmo Par-CSP

El Algoritmo 11, muestra el pseudocódigo para el Par-CSP. En la primera operación (línea 1), cada procesador cuenta las 1-secuencias de la porción de la base de datos que le fue asignada. Una posterior suma es realizada (línea 2) para calcular el soporte global, (que son almacenadas en la variable *Global_Counts* en cada procesador) e identificar las 1-secuencias frecuentes que serán almacenadas en la variable *F1.* Luego (línea 3) cada procesador

construye las pseudo proyecciones para las 1-secuencias frecuentes, dentro de la porción asignada de la base de datos. Las pseudo proyecciones son enviadas a todos los procesadores (línea 4). Antes de programar las proyecciones, Par-CSP aplica el muestreo selectivo para estimar el tiempo de minado relativo de estas proyecciones (línea 5). Estos tiempos de minado relativo son asignados a la variable *S_Result.* Después del muestreo, la proyección con el mayor de los tiempos, es particionada en otras más pequeñas (línea 6). Entonces el procesador máster programa estas proyecciones como subtareas contenidas en una cola de tareas siguiendo la estructura de anillo virtual mencionada anteriormente (línea 8-13).

Algoritmo 11: Par-CSP

Input: Identificador del procesador: *I,* porción del conjunto de datos asignada al procesador I :DB_I, soporte mínimo: *minsup*
Output: Porción de secuencias frecuentes cerradas: CSP_I

1 C_I = número de *1*-secuencias frecuentes (DB_I)
2 *Global_Counts* = *all_to_all_sum*(C_I) ; *F1* = *1*-secuencias frecuentes(*Global_Counts*);
3 *PSP*= pseudo proyección (*F1,* DB_I);
4 *Global_PSP* = *all_to_all broadcast*(*PSP*);
5 *S_Result* = muestreo selectivo(*F1, I,* DB_I, *minsup*);
6 *F2* = partición (*F1*, *S_Result*) // Particiona la proyección que consuma el mayor de los tiempos y asigna el nuevo conjunto de proyecciones a *F2*.
7 **if(***I* == 0**) then**
8 Acepta el pedido de los nodos esclavos y responde a cada pedido con un identificador diferente, desde el conjunto *F2* hasta que todas las proyecciones hayan sido asignadas.
9 **else**
10 envía pedido para un identificador de proyecciones al nodo maestro;
11 detener, si todas las proyecciones han sido asignadas;
12 aplica algoritmo BIDE a elementos de *Global_PSP* asignados por el procesador master;
13 Acumular las secuencias frecuentes cerradas en CSP_I y regresar a la operación de envío de pedidos.
14 **end if;**

6.4 Algoritmo paralelo basado en FP-tree

Recientemente en el año 2009, Wang,Chen y Zhou, tras realizar un estudio sobre los algoritmos para la minería de patrones secuenciales, presentan la implementación en paralelo, de un algoritmo serie basado en el crecimiento de patrones mediante bases de datos proyectadas y que utiliza una estructura de datos del tipoFP-tree, donde se expresan todas las secuencias posibles [25]. El algoritmo serie, a partir de las 1-secuencias frecuentes (como patrón sufijo inicial), construye su patrón de base condicional (FP-tree, que consiste de un conjunto de prefijos que ocurren junto con el patrón sufijo); creada esta condicional, el algoritmo mina recursivamente el árbol. El crecimiento de patrones es alcanzado mediante la concatenación de los patrones sufijos, con los patrones frecuentes generados a partir del FP-tree condicional.

La implementación en paralelo de este algoritmo se realizó sobre la arquitectura de una máquina de memoria distribuida, asumiendo que la transferencia de mensajes es la única vía para la comunicación entre procesadores, además se supone que la base de datos es demasiado grande para cargarla directamente en la memoria principal, esto hace que el algoritmo tenga mejor escalabilidad cuando se enfrenta a enormes conjuntos de datos [25].

Para este tipo de problemas los autores presentan dos enfoques paralelos: paralelismo de datos y paralelismo de tareas.

Aplicando paralelismo de datos

La idea del paralelismo de datos viene dada por la necesidad de calcular el soporte de diferentes nodos de secuencias entre diferentes procesadores. Siendo P el número de procesadores, se divide la base de datos original, en P partes de igual tamaño y cada bloque será asignado a un procesador diferente y almacenado en su disco local. Cada procesador calcula el soporte de su subconjunto local asignado y posteriormente en una operación de suma-reducción se calcula el soporte global, a partir del cual se determina si los nodos de secuencias cumplen la condición de soporte mínimo [25].

Aplicando paralelismo de tareas

Una vez que la base de datos de secuencias ha sido proyectada a diferentes nodos, el cálculo del soporte en cada nodo puede ser visto como una tarea independiente y asignarle cada tarea a un procesador diferente. Existen dos modelos que pueden ser utilizados para distribuir las tareas relacionadas a los diferentes nodos en el árbol: modelo horizontal y vertical. En el modelo horizontal, los nodos son generados por cada nivel en turno y las tareas, asignadas a estos nodos, serán distribuidas a diferentes procesadores con diferente descomposición en cada nivel. En el modelo vertical, cada procesador distribuye una serie de tareas a los nodos específicos del nivel k+1 y generan de forma independiente todos los sub-arboles enraizados en estos nodos. Para las computadoras con arquitectura de memoria distribuida el rendimiento del modelo vertical es superior al horizontal, de la misma forma que el modelo horizontal es más adecuado para sistemas de memoria compartida[25].

Estrategia de balanceo de carga

Con el objetivo de alcanzar un buen balanceo de carga entre los procesadores, los autores adoptan una estrategia de Balanceo de carga dinámico para la asignación de las tareas, basándose en un modelo aleatorio. En este modo cuando un procesador está desocupado envía, al azar, una solicitud de trabajo a otros procesadores. El procesador receptor dará una respuesta: si tiene trabajo extra asignará algún trabajo al procesador que solicita, de lo contrario los procesadores desocupados continuarán enviando la solicitud a otros procesadores [25].

7 Conclusiones y trabajo futuro

En el presente reporte se abordó el estado del arte, actualizado hasta el 2009, de los principales algoritmos para el minado de FCMS, así como las ventajas que ofrece la introducción de la computación paralela en los métodos de minería de secuencias, que aumentan la eficiencia y la escalabilidad de estos algoritmos.

Se describieron algoritmos series para el minado de secuencias frecuentes, agrupados en tres grandes grupos: los basados en heurísticas Apriori, en bases de datos con formato vertical y en crecimiento de patrones; además se analizó un nuevo algoritmo llamado MEMISP que introduce nuevas técnicas para el trabajo con memoria. Vimos que estos algoritmos minan todo el espacio de búsqueda de secuencias lo cual reduce la eficiencia y la eficacia de las tareas a ejecutar. En consecuencia analizamos también algoritmos que se caracterizaban por generar solamente FCS y se describieron estrategias más eficientes para podar el espacio de búsqueda.

También se describieron las implementaciones en paralelo de algunos de los algoritmos series expuestos. En este sentido se pudo concluir que el algoritmo N/S/HSPSM, diseñado para una máquina de memoria distribuida (MD), genera una gran cantidad de secuencias candidatas, característica que lo convierte en un algoritmo poco práctico. También se analizaron otros dos algoritmos paralelos llamados PSPADE y Par-CSP, que utilizan una estructuración del espacio de búsqueda basada sobre el enfoque del retículo/árbol de secuencias, aplicando el paralelismo de tareas mediante la partición de los nodos del nivel

superior. El primero mina todo el espacio de búsqueda de secuencias frecuentes, con arquitectura de memoria compartida (MC), y el segundo solo mina el espacio de búsqueda de FCS, diseñada también para una máquina de memoria distribuida, ver Figura 14.También se analizó un estudio reciente sobre algoritmos paralelos en la minería de patrones secuenciales que utilizan estructuras de datos tipo árbol (*FP-tree*), y se describieron algunas de las estrategias utilizadas para implantar el paralelismo de datos y el paralelismo de tareas.

Basado en el análisis realizado, existen algunas líneas de trabajo que pudieran ser tratadas en trabajos futuros:

- Incorporación de restricciones de espacio en la minería de secuencias frecuentes con el objetivo de enfocar la búsqueda sobre patrones secuenciales interesantes.
- Lograr una mayor compresión en el conjunto final de FS, utilizando métodos de búsqueda para secuencias frecuentes cerradas y maximales.
- Implementación de algoritmos paralelos para la minería de secuencias frecuentes cerradas y maximales, sobre dispositivos de aceleración por hardware (FPGA, GPU, etc.), o utilizando una arquitectura de Clusters, para lograr un mayor rendimiento y escalabilidad de los mismos.

Referencias bibliográficas

1. B. A. Davey, H. A. Priestley. Introduction to lattices and order. Cambridge: Cambridge University Press, 1990.
2. R. Agrawal, R. Srikant. Fast Algorithms for mining association rules. In: Proceeding of the VLDB Conference. Santiago de Chile, September1994.
3. R. Agrawal, R. Srikant. Mining sequential patterns. IEEE Computer Society: In Proc. of the 11th International Conference on Data Engineering (ICDE' 95). Taipei, Taiwan, March, 1995.
4. R Srikant, Agrawal R. Mining sequential patterns: Generalizations and performance improvements. In: 5th Intl. Conf. Extending Database Technology, 1996.
5. A. A. Freitas, S. H. Lavington. Mining Very Large Databases with Parallel Processing. Kluwer Academic Publishers. Norwell, MA, 1997.
6. R.J. Bayardo.Efficiently mining long patterns from databases. In Proc. 1998 ACM-SIGMOD Int.Conf. Management of Data (SIGMOD'98), Seattle, WA. June 1998.
7. T. Shintani, M. Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In: Proc. of the Second Pacific-Asia Conference on Research and Development in Knowledge Discovery and Data Mining. Melbourne, Australia, 1998.
8. F. Provost, V.Kolluri. A survey of methods for scaling up inductive Algorithms. Data Mining and Knowledge Discovery, 1999.
9. J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, M.-C. Hsu. FreeSpan: Frequent pattern-projected sequential pattern mining. In Proc. 2000 Int. Conf. Knowledge Discovery and Data Mining, KDD'00. Boston, MA, Aug. 2000.
10. J. Pei, J. Han, R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In Proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'00). Dallas, TX, May 2000.
11. M. J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. Journal Machine Learning, 2001.
12. J.Pei, J. Han,H.Pinto, Q.Chen, U. Dayal, M. C. Hsu. PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In Proc. of 2001 International Conference on Data Engineering, 2001.
13. D. Burdick, M. Calimlim, J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In Proc. 2001 Int. Conf. Data Engineering (ICDE'01), Heidelberg, Germany, April 2001.
14. M. J.Zaki.Parallel sequence mining on shared-memory machines. Journal on Parallel Distributed Computing, 2001.
15. M. J. Zaki, C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. : In Proc. 2002 SIAM Int. Conf. Data Mining (SDM'02). Arlington, VA, April 2002.
16. V. Kumar. High Performance Data Mining. 2002.

17. R.Afshar.Mining Frequent Max and Closed Sequential Patterns. May, 2002.
18. X. Yan, J. Han, R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In SDM'03, 2003.
19. J. Wang, J. Han.BIDE efficient mining of frequent closed sequences. In ICDE'04, 2004.
20. L.M.Lee, L. S. Y.Yen.Fast discovery of sequential patterns through memory indexing and database partitioning. Journal Information Science and Engineering, vol. 21. 2005.
21. S. Cong, J. Han, D. Padua. Parallel mining of closed sequential patterns. In Proc. of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining. Chicago, USA, 2005.
22. J. Dongorra, I. Foster. Sourcebook of parallel computing. Elsevier Science, 2003.
23. M. Chen, J. Han, P. Yu. Data mining: an overview from a database perspective. Ieee Trans. On Knowledge and Data Engineering, 1996.
24. G. Karypis, V. Kumar, A. Gupta, A. Grama. Introduction to Parallel Computing. Second Edition, 2003.
25. J. Wang, X. Chen, K. Zhou. Research on a Scalable Parallel Data Mining Algorithm. Fifth International Joint Conference on INC. Seoul, Korea, August, 2009.
26. J. H. Palancar. Estado actual de la aplicación de la computación paralela a la Minería. Reporte Técnico. La Habana, Cuba. Noviembre, 2004.

RT_013, abril 2010
Aprobado por el Consejo Científico CENATAV

Editor: Lic. Lucía González Bayona
Diseño de Portada: DCG Matilde Galindo Sánchez
RNPS No. 2143
ISSN 2072-6260
Indicaciones para los Autores:
Seguir la plantilla que aparece en www.cenatav.co.cu
C E N A T A V
7ma. No. 21812 e/218 y 222, Rpto. Siboney, Playa;
Ciudad de La Habana. Cuba. C.P. 12200
Impreso en Cuba