RNPS No. 2143
ISSN 2072-6260
Versión Digital

SERIE GRIS

REPORTE TÉCNICO
Minería de Datos

Descubrimiento de conjuntos frecuentes de Ítems en datos estáticos y dinámicos

MSc. Raudel Hernández León, Dr. C. José Hernández Palancar, Dr. C. Jesús A. Carrasco Ochoa y Dr. C. José Fco. Martínez Trinidad

RT_006 **agosto 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

Descubrimiento de conjuntos frecuentes de Ítems en datos estáticos y dinámicos

MSc. Raudel Hernández León, Dr. C. José Hernández Palancar, Dr. C. Jesús A. Carrasco Ochoa y Dr. C. José Fco. Martínez Trinidad

RT_006 **agosto 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 general

Índice de figuras

Descubrimiento de conjuntos frecuentes de ítems en datos estáticos y dinámicos

MSc. Raudel Hernández León, Dr. C. José Hernández Palancar, Dr. C. Jesús A. Carrasco Ochoa y Dr. C. José Fco. Martínez Trinidad

Centro de Aplicaciones de Tecnología de Avanzada (CENATAV),
7a ♯ 21812 e/ 218 y 222, Rpto. Siboney, Playa, C.P. 12200, La Habana, Cuba.
rhernandez@cenatav.co.cu

RT_006 CENATAV
Fecha del camera ready: 5 de mayo de 2009

Resumen: La cantidad de datos que se genera hoy en día en cualquier área de conocimiento rebasa la capacidad de asimilación de cualquier ser humano. Entre las técnicas de minería de datos más utilizadas se encuentra el descubrimiento o minado de conjuntos frecuentes de ítems (FI). El minado de FI ha sido aplicado en clasificación y agrupamiento de documentos, en análisis de información de ventas, en telecomunicaciones, etc. En este reporte técnico se presenta un estado del arte actualizado hasta julio de 2008, así como dos algoritmos para minar todos los conjuntos frecuentes de ítems en grandes volúmenes de datos. El primero de los algoritmos (*CA*) procesa conjuntos de datos estáticos, es decir, que no cambian. *CA* combina la representación binaria con una estructuración del espacio de búsqueda en clases de equivalencia para reducir el consumo de memoria y realizar un rápido cálculo de los FI. El segundo algoritmo presentado en este reporte se denomina *ICA* y procesa conjuntos de datos dinámicos, datos en los que se puede adicionar, eliminar o modificar la información existente.

Palabras clave: minería de datos, reglas de asociación, patrones frecuentes

Abstract: Currently, the amount of data generated in any knowledge area is too big therefore this amount of data cannot be processed by a human. Mining or discovering frequent items is one of the most used data mining techniques which has been applied for classification, document clustering, data analysis in marketing and communications, etc. In this technical report, a state of the art until July 2008 as well as two algorithms for mining frequent itemsets (FI) on big sparse datasets are presented. The first algorithm named Compressed Arrays (*CA*) processes static data, i.e. datasets which do not change. CA performs a breadth first search through equivalence classes and introduces compressed arrays to accumulate the prefix class supports. The second algorithm named Incremental Compressed Arrays (*ICA*) processes dynamic data, i.e. datasets in which a set of transactions can be added, deleted or modified.

Keywords: Data Mining, Association Rules, Frequent Patterns

1. Introducción

La cantidad de datos que se genera y almacena hoy en día en cualquier área de conocimiento es tan vasta, que rebasa la capacidad de asimilación de cualquier ser humano. Muchas son las técnicas de minería de datos desarrolladas para tratar este problema, e.g. clasificación, agrupamiento, descubrimiento o minado de patrones frecuentes, reglas de asociación, etc. Estas técnicas, de una forma u otra, extraen conocimiento potencialmente útil y comprensible, a partir de grandes volúmenes de datos. A lo largo de este reporte, al hablar de

conjunto de datos estaremos haciendo referencia a datos transaccionales, los cuales pueden verse como un archivo texto donde cada fila (transacción) está compuesta por un conjunto de elementos o ítems.

El presente trabajo aborda el minado de patrones frecuentes, en particular el minado de conjuntos frecuentes de ítems (FI por sus siglas en inglés). En la literatura existen trabajos enfocados a la obtención de secuencias frecuentes de ítems [16,17]. Debido a que pueden confundirse los conceptos conjunto de ítems y secuencia de ítems, a continuación se mencionan algunas diferencias entre ambos conceptos:

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

El minado de FI tiene diversas aplicaciones como: soporte para toma de decisiones, predicción y diagnóstico de alarmas en telecomunicaciones, análisis de información de ventas [8,13] entre otras.

Los algoritmos desarrollados para el minado de FI, al tratar con conjuntos de datos muy grandes y dispersos[1], por ejemplo las colecciones de documentos, tardan mucho en dar respuesta o simplemente no pueden procesarlos. Al decir conjuntos de datos muy grandes nos referimos a conjuntos de datos con millones de ítems diferentes y/o millones de transacciones. La demora de los algoritmos existentes para procesar estos conjuntos se debe, fundamentalmente, a la presencia de miles de ítems diferentes (en algunos casos millones) y/o al uso de un umbral de soporte[2] muy bajo.

De forma general, existen dos estrategias para recorrer el espacio de búsqueda de los ítems frecuentes: en amplitud y en profundidad. Los algoritmos que siguen una estrategia en amplitud necesitan generar todos los conjuntos de k ítems antes de generar los conjuntos de $k+1$ ítems. Si la cantidad de ítems es muy grande o el umbral de soporte (en adelante *minsup*) es muy bajo se generan muchos conjuntos candidatos a ser frecuentes, lo cual puede ser imposible de mantener en memoria. Es importante notar que si se tienen n ítems diferentes y un *minsup* cercano a cero, la cantidad de FI tiende a 2^n.

Los algoritmos que siguen una estrategia en profundidad utilizan estructuras arbóreas para almacenar todo el conjunto de datos en memoria; si los conjuntos de datos son muy grandes, las estructuras arbóreas resultantes son grandes también y los recorridos sobre estas son muy costosos en tiempo.

Además de que los conjuntos de datos pueden ser muy grandes, existen problemas donde los conjuntos de datos son dinámicos y en éstos se puede adicionar, eliminar o modificar transacciones. El tratamiento de datos dinámicos es imprescindible en los sistemas que, además de manejar un gran volúmen de información, necesitan actualizar la información contantemente. Si se tiene un conjunto de datos muy grande, no es conveniente, después de cada actualización, recorrerlo completamente para recalcular los FI. En general, son tres los problemas que se presentan en un proceso dinámico de minado de FI:

[1] Los conjuntos de datos dispersos tienen un promedio de ítems por transacción muy pequeño, por debajo del 30 %, con respecto al total de ítems.

[2] Se denomina umbral de soporte al por ciento o cantidad mínima de ocurrencias que necesita un conjunto de ítems para ser considerado frecuente.

1. Variar el *minsup* sin actualizar los datos.
2. Adicionar, eliminar o modificar transacciones manteniendo un *minsup* invariable.
3. Adicionar, eliminar o modificar transacciones con un *minsup* variable.

La mayoría de los algoritmos desarrollados para datos dinámicos se centra en adicionar y una pequeña parte de esos algoritmos en eliminar o modificar transacciones con un *minsup* invariable. Es importante señalar que, al modificar una o varias transacciones, algunos FI pueden dejar de serlo y peor aún, otros conjuntos de ítems que no habían sido frecuentes hasta algún momento pueden volverse frecuentes.

El reporte consta de 6 secciones. En la sección 2 se presentan algunos conceptos básicos necesarios a lo largo del documento. En la seccíon 3 se aborda el trabajo relacionado tanto para datos estáticos como para datos dinámicos. En las secciones 4 y 5 se describen los dos algoritmos presentados en este reporte y en la sección 6 se dan las conclusiones, así como el trabajo futuro.

2. Conceptos básicos

A continuación se presentan algunos conceptos necesarios para el resto del reporte.

Sea $I = \{i_1, i_2, \ldots, i_n\}$ un conjunto de n ítems. Sea D un conjunto de transacciones, donde cada transacción T es un conjunto de ítems tal que $T \subseteq I$. Cuando se haga referencia a un conjunto de ítems X, se estará hablando de un subconjunto de I y se asumirá, sin perder generalidad, que existe un orden lexicográfico entre los ítems. El soporte de un conjunto de ítems X es el número de transacciones en D que contienen a X. Si el soporte de un conjunto de ítems es mayor o igual que un umbral de soporte previamente establecido, entonces el conjunto de ítems se denomina frecuente. Véase por ejemplo la tabla 1. Si se

Tabla 1. Conjunto de datos transaccionales

id	ítems
1	coca, leche
2	pan, pañal, cerveza
3	coca, pañal, cerveza
4	pan, pañal, cerveza
5	coca, leche, pañal

establece como *minsup* 60 % ($0{,}6 * 5 \equiv 3$ ocurrencias como mínimo para ser considerado frecuente) se obtienen los FI siguientes: {coca}, {pañal}, {cerveza} y {pañal, cerveza}, dado que son los únicos que ocurren 3 o más veces. El tamaño de un conjunto de ítems se define por su cardinalidad, un conjunto de ítems compuesto por k ítems se denomina *k-itemset*.

La mayoría de los algoritmos para el minado de FI se basan en el algoritmo *Apriori* introducido por Agrawal *et al.* [1]. El mismo Agrawal formuló en [2] la propiedad de clausura descendente del soporte, la cual plantea que todo subconjunto de un FI es frecuente o lo que es igual, todo superconjunto de un conjunto no frecuente es no frecuente. Esta propiedad permite podar el espacio de búsqueda evitando calcular el soporte de conjuntos de ítems que no pueden ser frecuentes.

2.1. Representación de los datos

Un factor que incide decisivamente en el cálculo del soporte de los conjuntos de ítems es la forma en que se representen los datos. Conceptualmente, un conjunto de datos transaccionales es una matriz de dos dimensiones, donde las filas representan las transacciones y las columnas representan los ítems diferentes. En la literatura se han reportado cuatro formas diferentes de almacenar esta matriz [33]:

- Lista Horizontal de Ítems (*LHI*): Se almacena por cada transacción una lista con los ítems presentes en ella (Fig. 2.1).
- Vector Horizontal de Ítems (*VHI*): Se almacena por cada transacción un vector binario donde un 1 en la posición i-ésima denota la presencia del i-ésimo ítem en la transacción y un 0 denota la ausencia (Fig. 2.1).
- Lista Vertical de Identificadores de Transacciones (*LVTid*): Se almacena por cada ítem una lista con los identificadores de las transacciones donde aparezca el mismo (Fig. 2.1).
- Vector Vertical de Identificadores de Transacciones (*VVTid*): Se almacena por cada ítem un vector binario donde un 1 en la posición i-ésima denota la presencia del ítem en la i-ésima transacción y un 0 denota la ausencia (Fig. 2.1).

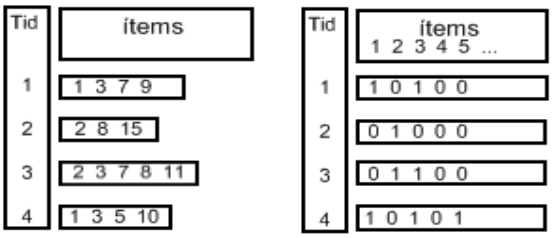

Fig. 1. Lista horizontal y vector horizontal de ítems

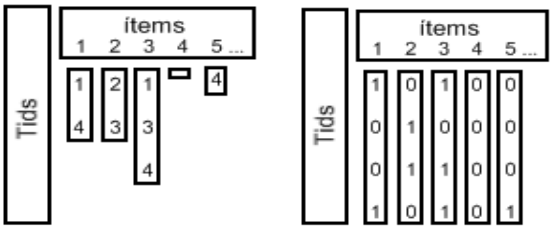

Fig. 2. Lista vertical y vector vertical de tids

Gran parte de los algoritmos reportados utilizan las representaciones basadas en listas, tanto horizontales como verticales, con la idea de que las listas consumen menos memoria que los vectores binarios. Esta diferencia en el consumo de memoria se evidencia más en los conjuntos de datos dispersos, ya que las listas no tienen el sobrecosto de la representación explícita de la ausencia de ítems. Las representaciones verticales, a pesar de ser las menos empleadas, facilitan el cálculo del soporte que se puede obtener intersectando las listas o vectores asociados a los ítems. Los dos algoritmos presentados en este trabajo almacenan el conjunto de datos en el formato *VVTid*.

2.2. Recorridos en el espacio de búsqueda

No sólo existe diferencia en la forma en que se representan los datos sino también en la forma en que se recorre el espacio de búsqueda de los ítems frecuentes. El cálculo de todos los FI es una tarea que consume muchos recursos, debido a su complejidad exponencial y por lo general, se corresponde con un recorrido en el espacio de búsqueda asociado al retículo[3] formado por los conjuntos de ítems [39,18]. Para comprender mejor lo anterior, supóngase un conjunto universo de cuatro ítems: $I = \{i_1, i_2, i_3, i_4\}$. El retículo, omitiendo el conjunto vacío, que forman los subconjuntos del conjunto I, puede observarse en la figura 3. Es fácil comprobar que los soportes de los conjuntos de ítems que forman el retículo

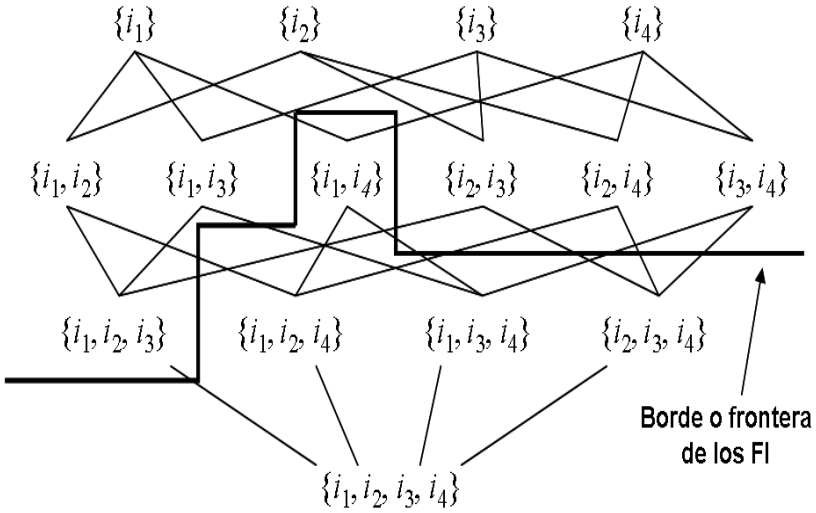

Fig. 3. Retículo formado por los conjuntos de ítems

disminuyen al recorrerlos desde los 1-*itemsets* hasta el conjunto universo, verificándose la clausura descendente del soporte. Por tanto, cualquiera que sea el *minsup*, todo retículo posee un borde o frontera que separa los FI del resto de los conjuntos de ítems (Fig. 3).

Los algoritmos de minado de FI definen diferentes estrategias para recorrer el espacio de búsqueda. Las estrategias pueden clasificarse, atendiendo a la dirección de los recorridos, en:

- Descendentes: Si el recorrido se realiza desde los 1-*itemsets* hasta la frontera.
- Ascendentes: Si el recorrido se realiza en el sentido opuesto, desde superconjuntos de los conjuntos de ítems que forman la frontera hasta la frontera.

Al mismo tiempo, dentro de estas estrategias se pueden generar los conjuntos de ítems de dos formas:

[3] En matemática, un retículo, red o *lattice* es un conjunto parcialmente ordenado en el cual todo subconjunto finito no vacío tiene un supremo y un ínfimo.

1. En amplitud (*Breadth-First Search*): Se generan todos los *k-itemsets* frecuentes antes de generar los $(k+1)$-*itemsets*. Algunos de los algoritmos que siguen esta estrategia son referidos en [2,31,29,20,21,19,22]. De esta forma, se emplea la propiedad de clausura descendente para podar los $(k+1)$-*itemsets* que tengan al menos un subconjunto no frecuente de tamaño k.
2. En profundidad (*Depth-First Search*): Se generan los conjuntos por cada rama de la estructura arbórea que se deriva del retículo (Fig. 4), es decir, se toma el primer ítem y se extiende con todos los ítems lexicográficamente mayores que él, tanto como sea posible, generando todos sus superconjuntos frecuentes y haciendo retroceso (*backtracking*) cuando no es posible extender más. Después de terminar con un ítem, se toma el siguiente y se realiza el mismo proceso. Algunos de los algoritmos que siguen esta estrategia son referidos en [38,39,15,4,30,34,14].

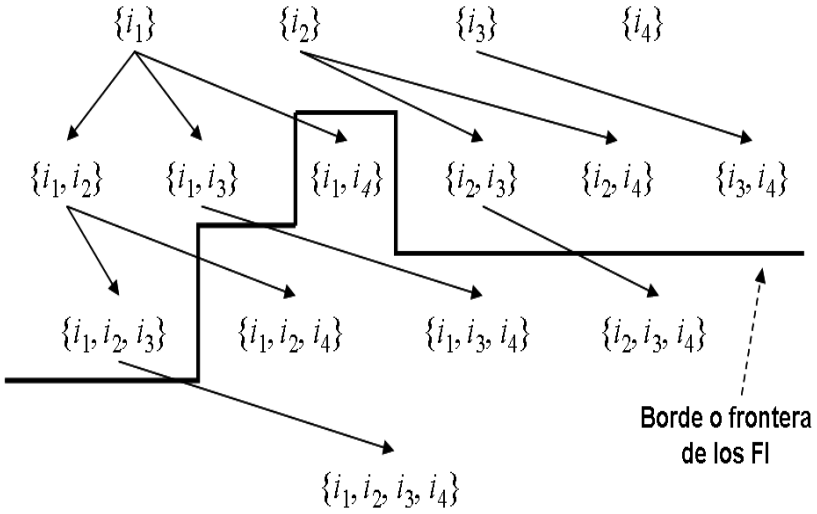

Fig. 4. Estructura arbórea derivada del retículo

Los dos algoritmos presentados en este trabajo utilizan estrategias de recorrido descendentes y generan los conjuntos de ítems en amplitud.

2.3. Estructuración del espacio de búsqueda

Si se observa la estructura arbórea derivada del retículo (Fig. 4), se puede apreciar que el conjunto de ítems representado en cada nodo es el resultado de la extensión, con un nuevo ítem, del conjunto de ítems representado en el nodo padre; en el caso de los nodos raíz de los árboles, se puede asumir al conjunto vacío como su padre. Además, los conjuntos de ítems representados en los hijos de un nodo coinciden en todos sus ítems menos en el último. Por tanto, el árbol cuyo nodo raíz contiene el primer ítem abarca la mitad del espacio de búsqueda; el árbol cuyo nodo raíz contiene el segundo ítem abarca la cuarta parte y así sucesivamente (Fig. 5).

En [39], el autor agrupa los conjuntos de ítems de longitud k que coinciden en sus primeros $k-1$ ítems (*prefijo*) en clases de equivalencia. En la figura 5 se muestran, con línea gruesa, las clases de equivalencia de los 1-*itemsets* $\{i_1\}$, $\{i_2\}$ e $\{i_3\}$ en ese orden. El ítem i_4 es el mayor lexicográficamente, por lo que no es prefijo de ninguna clase de equivalencia aunque sí puede pertenecer a las clases de equivalencia anteriores. En el segundo nivel y con línea discontinua se señala la clase de equivalencia del prefijo $\{i_1, i_2\}$. No se señalan las demás clases de equivalencia para no sobrecargar la figura. El tamaño de una clase de

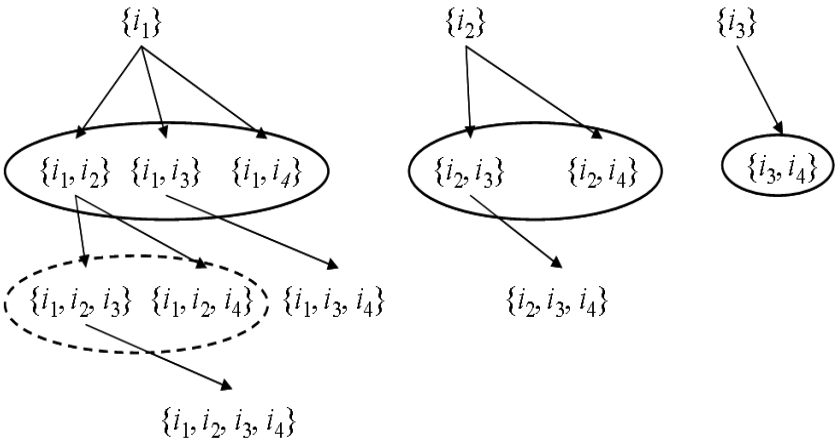

Fig. 5. Clases de equivalencia presentes en el retículo

equivalencia está dado por la cardinalidad de los conjuntos de ítems que ésta agrupa. Como se verá más adelante en el reporte, las características de las clases de equivalencia pueden ser muy útiles en el proceso de minado de los FI. Ambos algoritmos presentados (*CA* e *ICA*) realizan una estrategia de recorrido en amplitud por cada clase de equivalencia.

3. Trabajo relacionado

Los primeros trabajos relacionados con el minado de FI datan de principios de la década de los 90. En esta sección se ha hecho una selección de los algoritmos que consideramos más importantes, ya sea por su eficiencia como por su aporte teórico. En todos los casos se presentarán sus ventajas y desventajas sin profundizar en los detalles de implementación, los cuales no siempre son descritos por los autores.

3.1. Algoritmos para datos estáticos

A continuación se presentan los algoritmos para conjuntos de datos estáticos, que no cambian, entiéndase por "algoritmo para conjunto de datos estáticos" un algoritmo que supone que los datos no van a cambiar y si cambiaran, para actualizar el conjunto de FI se debe ejecutar nuevamente el algoritmo sobre todo el conjunto de datos.

Los primeros algoritmos presentados realizan generación de candidatos, es decir, generan todos los *k-itemsets* frecuentes y los utilizan para generar y evaluar candidatos a ser

(k+1)-*itemsets* frecuentes. Seguidamente se analizan algoritmos que no generan candidatos. Estos algoritmos, por lo general, almacenan el conjunto de datos en estructuras arbóreas y las recorren siguiendo una estrategia en profundidad utilizando diferentes heurísticas. Por último, se describen dos algoritmos que siguen una estrategia de recorrido ascendente.

Hay algunos trabajos que usan vectores verticales de identificadores de transacciones (*VVTid*) para almacenar la matriz de datos [33,5,28], pero no son descritos por no ser muy eficientes los algoritmos presentados en esos trabajos. En este reporte se presentan dos algoritmos que usan precisamente este tipo de almacenamiento (secciones 4 y 5).

Apriori y sus variaciones

De todos los algoritmos existentes en la literatura, *Apriori* ha sido uno de los de mayor impacto y posiblemente el más referenciado [2].

El algoritmo *Apriori* sigue una estrategia de recorrido en amplitud, realiza un almacenamiento utilizando listas horizontales de ítems, *LHI*, de la matriz de datos y genera iterativamente dos conjuntos: C_k y L_k. El conjunto L_k contiene los k-*itemsets* frecuentes y el conjuntos C_k contiene los k-*itemsets* candidatos, los cuales representan un superconjunto de L_k.

El conjunto L_k se obtiene recorriendo el conjunto de datos y calculando el soporte de cada k-itemset candidato en C_k. El conjunto C_k se genera a partir del conjunto L_{k-1}, como se indica en 1:

$$C_k = \{c \mid \mathit{Unión}(c, L_{k-1}) \wedge \mathit{Poda}(c, L_{k-1})\} \tag{1}$$

donde:

$$\mathit{Unión}(\{i_1, i_2, \ldots, i_{k-2}, i_{k-1}, i_k\}, L_{k-1}) \equiv$$
$$\langle \{i_1, \ldots, i_{k-2}, i_{k-1}\} \in L_{k-1} \wedge \{i_1, \ldots, i_{k-2}, i_k\} \in L_{k-1} \rangle, \tag{2}$$

$$\mathit{Poda}(c, L_{k-1}) \equiv \langle \forall s[(s \subset c \wedge |s| = k-1) \rightarrow s \in L_{k-1}] \rangle \tag{3}$$

En la expresión 1 se realiza una *unión* sobre los $k-2$ primeros ítems. Esta operación consiste en tomar todos los pares de $(k-1)$-*itemsets* que coincidan en sus $k-2$ primeros ítems y generar un k-*itemset* manteniendo los $k-2$ ítems comunes y adicionando, en orden lexicográfico, los $(k-1)$-ésimos ítems de los dos conjuntos que se unen (2). Después de realizar la unión, se aplica la propiedad de clausura descendente del soporte para podar los k-*itemsets* que tengan al menos un subconjunto no frecuente de tamaño $k-1$ (3).

A continuación se presenta el pseudocódigo del algoritmo *Apriori*:

En la línea 1 se calculan los 1-*itemsets* frecuentes, de la línea 2 a la 8 se generan los FI nivel por nivel. En la línea 3 se generan los conjuntos candidatos de tamaño k y de la línea 4 a la 7 se recorre el conjunto de datos transacción por transacción generando los k-*itemsets* de cada una e incrementando sus soportes en caso de pertenecer a C_k. En la línea 8 se verifican los soportes eliminando los k-*itemsets* no frecuentes.

Para lograr un eficiente cálculo del soporte del conjunto C_k se construye un *hash-tree* por cada nivel, éste permite verificar rápidamente la existencia o no, en el conjunto C_k, de los conjuntos generados en cada transacción. Para ver en detalle la estructura *hash-tree*, consúltese [2]. *Apriori* utiliza la propiedad de clausura descendente del soporte para podar el espacio de búsqueda, pero presenta claras desventajas como la necesidad de mantener

Algoritmo 3.1: Apriori

Input: conjunto de datos transaccionales T,
umbral de soporte *minsup*
Output: conjuntos frecuentes de ítems F

```
1  L_1 = {1-itemsets frecuentes};
2  for (k = 2; L_{k-1} ≠ ∅; k++) do
3      C_k = {c | Unión(c, L_{k-1}) ∧ Poda(c, L_{k-1})};
4      forall transacción t ∈ T do
5          C_t = {c ⊆ t | c ∈ C_k};
6          forall candidato c ∈ C_t do
7              c.soporte++;
8          end
9      end
10     L_k = {c ∈ C_k | c.soporte ≥ minsup};
11 end
12 F = ⋃_k L_k
```

todo un nivel en memoria para generar los candidatos del siguiente nivel y además, realiza múltiples recorridos por el conjunto de datos.

En [2] se presentan además dos variaciones del algoritmo *Apriori*, la primera se denomina *AprioriTid* y la segunda *AprioriHybrid*.

En *AprioriTid*, los conjuntos de ítems que se van generando se almacenan en pares o registros de la forma $\langle Tid, \{X_k\}\rangle$, siendo X_k el conjunto de los *k-itemsets* candidatos presentes en la transacción Tid; es importante notar que, en todo momento, la cantidad de registros es menor o igual que la cantidad de transacciones y, al aumentar k, tiende a disminuir la cantidad de registros. En cada iteración, en vez de recorrer las transacciones como en el algoritmo *Apriori*, se recorren los registros que resultan de la iteración anterior.

Esta variante tiene la ventaja de eliminar los registros donde no se generen nuevos candidatos; además, cuando k es grande, cada registro tiende a ser menor que la transacción original. Por otro lado, cuando k es pequeño, los registros tienden a ser mayores que las transacciones porque se generan muchos candidatos por transacción. El algoritmo *AprioriTid* sólo es más eficiente que el algoritmo *Apriori* cuando el conjunto de registros puede almacenarse en memoria y además, la densidad de FI por transacción es pequeña.

La variante *AprioriHybrid* combina los algoritmos *Apriori* y *AprioriTid*, aprovechando la mayor eficiencia del primero en los primeros recorridos por el conjunto de datos y la eficiencia del segundo, a partir de cierto k, cuando el conjunto de registros puede almacenarse en la memoria. Es posible estimar cuando el conjunto de registros cabe en memoria porque en cada nivel ambos algoritmos generan los mismos conjuntos candidatos.

El algoritmo para datos estáticos presentado en este reporte (CA) sólo recorre dos veces el conjunto de datos, mejorando así los algoritmos tipo *Apriori*.

Algoritmos de particionamiento

La mayor deficiencia de *Apriori* y sus variaciones es la necesidad de realizar múltiples recorridos por el conjunto de datos. Algunos trabajos eliminan esta deficiencia particionando el conjunto de transacciones [31].

Otros algoritmos, en vez de particionar el conjunto de transacciones, particionan el conjunto de ítems [21]. Esta estrategia es conveniente cuando la cantidad de ítems diferentes es muy grande con respecto a la cantidad de transacciones.

En [31] se presenta un algoritmo llamado *Partition* el cual divide el conjunto de transacciones en varias particiones disjuntas, no necesariamente de igual tamaño, de forma tal que cada partición por separado pueda almacenarse en memoria. Este algoritmo, a diferencia de *Apriori* y sus variaciones, recorre el conjunto de datos sólo dos veces. En el primer recorrido, se toma cada partición y se procesa independientemente para encontrar los FI locales de todos los tamaños. Seguidamente, se mezclan los resultados de cada partición combinando los FI de igual tamaño para generar el conjunto global de candidatos. En el segundo recorrido se calculan los soportes del conjunto global de candidatos para obtener los FI.

Dado que el umbral de soporte es un porcentaje del total de transacciones, si un conjunto de ítems no es localmente frecuente en ninguna partición, entonces no será frecuente globalmente. Por tanto, los conjuntos de ítems que no sean localmente frecuentes en ninguna partición no se consideran como candidatos globales, pudiendo existir falsos positivos pero nunca falsos negativos.

El algoritmo *Partition* logra reducir la cantidad de recorridos por el conjunto de datos a sólo dos; no obstante, la distribución de los ítems por cada transacción puede generar un conjunto muy grande de candidatos locales y globales, y como consecuencia, un segundo recorrido muy costoso. La lógica de este algoritmo ha sido aplicada en diversos algoritmos paralelos para calcular FI.

Algunos trabajos han sido enfocados al minado de FI en datos textuales, en estos datos la cantidad de ítems es muy grande y en ocasiones mayor que la cantidad de transacciones. En [21] se presentan los algoritmos *M-Apriori* y *M-DHP* (la *M* significa *Multipass*), ambos particionan el conjunto de ítems y proceden de la siguiente forma:

1. Se calcula el soporte de cada ítem para encontrar los 1-*itemsets* frecuentes.
2. Se particiona el conjunto de los 1-*itemsets* frecuentes en p particiones: $\{P_1, P_2, \ldots, P_p\}$.
3. Se ejecuta el algoritmo *Apriori* o el algoritmo *DHP* (ver sección 3.1) para encontrar todos los FI desde la última partición (P_p) hasta la primera partición (P_1). La heurística de este paso es la siguiente: Cuando se procesa la partición P_p, se encuentran todos los FI cuyos ítems están en P_p; cuando se procesa la próxima partición P_{p-1}, se encuentran todos los FI cuyos ítems están en P_{p-1} y P_p y se continúa de esta forma hasta que se procese P_1.

Ambos algoritmos asumen que existe un orden lexicográfico en los ítems, de forma tal que se cumple, para las p particiones, la siguiente restricción:

$$\forall_{i<j}, \forall_{a \in P_i}, \forall_{b \in P_j} (a < b) \tag{4}$$

Procesar las particiones, desde la última hasta la primera, permite disminuir la cantidad de conjuntos candidatos (ver en [21] las comparaciones con *Apriori* y *DHP*). Esto se debe a que cada partición se extiende con los FI resultantes de procesar las particiones posteriores a ella. El total de recorridos por el conjunto de datos que realizan, tanto *M-Apriori* como *M-DHP*, depende del número de particiones en que se dividan los 1-*itemsets* frecuentes.

Aunque estos algoritmos suelen ser eficientes en datos dispersos, presentan el problema de realizar tantos recorridos como cantidad de particiones se tenga y al procesar cada partición, necesitan generar los conjuntos candidatos de todos los tamaños. Contar al mismo tiempo los conjuntos candidatos de todos los tamaños resulta costoso, principalmente en las primeras particiones, las cuales generan el mayor número de candidatos.

Los algoritmos propuestos en este reporte, en vez de particionar el conjunto de ítems, particionan el espacio de búsqueda en clases de equivalencia y ordenan los 1-*itemsets* por orden ascendente del soporte. De esta forma, reducen el número de conjuntos candidatos en cada clase de equivalencia.

Algoritmos de *hashing*

Como se puede observar en los algoritmos presentados en la sección anterior, disminuir el número de candidatos generados influye positivamente en sus tiempos de ejecución. Algunos trabajos utilizan tablas *hash* para generar solamente conjuntos candidatos con una alta probabilidad de ser frecuentes. A continuación se presentan los algoritmos *DHP*, *IHP* y *MIHP* presentados en [29,22,19].

Además de la generación de grandes conjuntos candidatos de ítems, otro factor que afecta la eficiencia del minado de los FI es la cantidad de datos que se necesita recorrer para determinar el soporte de los mismos. En [29] se presenta un algoritmo denominado *DHP* (*Direct Hashing and Pruning*, por sus siglas en inglés) que utiliza una tabla *hash* para reducir la cantidad de conjuntos candidatos, podando aquellos que, con base en la propiedad de clausura descendente del soporte, no pueden ser frecuentes. El algoritmo *DHP* aplica también una heurística para reducir el tamaño del conjunto de datos, eliminando ítems y/o transacciones que dejen de aportar en el cálculo de soporte.

Al recorrer el conjunto de datos para calcular el soporte de los candidatos de tamaño k se guarda, por adelantado, información sobre cada posible candidato de tamaño $k+1$. Esta información se almacena en una tabla *hash* H_k asociada al nivel k. En cada celda de H_k se almacena un número que representa el soporte total de los $(k+1)$-*itemsets* con iguales valores de la función *hash* (la función *hash* no tiene por qué ser única). Es importante notar que, debido a las colisiones[4] que se presentan en las tablas *hash*, diferentes conjuntos de ítems pueden contarse en una misma celda. Por lo tanto, el valor de cualquier celda será la suma de los soportes de todos los conjuntos de ítems con iguales valores de la función *hash*. Luego, los conjuntos de ítems cuyos valores en la tabla *hash* no superen el *minsup* son excluidos del conjunto de candidatos del próximo nivel.

Para reducir la dimensión del conjunto de datos, *DHP* propone una heurística que se basa en la propiedad de clausura descendente del soporte. Esta heurística plantea que un ítem puede ser eliminado de una transacción si:

- No aparece en al menos k de los k-*itemsets* candidatos contenidos en la transacción.
- No aparece en al menos un $(k+1)$-*itemset* elegible de la transacción, considerando como elegible aquel que tiene todos sus $k+1$ conjuntos de ítems de tamaño k contenidos en el conjunto de candidatos C_k.

[4] Diferentes itemsets pueden tener el mismo valor de la función de *hash*.

De igual forma, una transacción completa puede eliminarse si no contiene al menos $k+1$ conjuntos de ítems de tamaño k en el conjunto de candidatos C_k.

El algoritmo *DHP* es más eficiente que los algoritmos previos a él, pero aún tiene la dificultad de realizar varios recorridos por el conjunto de datos, sólo evita algunos recorridos contando en un recorrido los candidatos de dos o tres niveles anteriores según pueda almacenarlos en memoria. Otro problema de los algoritmos que usan tablas *hash* es que cuando el *minsup* es muy pequeño, la cantidad de colisiones se hace muy grande y por tanto, la poda pierde efectividad. Si para evitar las colisiones se utilizara una tabla *hash* demasiado grande, se estaría haciendo lo mismo que en el algoritmo *Apriori* e incluso, sería menos eficiente, porque no se podría aplicar la clausura descendente del soporte para podar.

En trabajos como [22] y [19], en vez de acumular el soporte de los conjuntos de ítems en tablas *hash*, se acumulan los identificadores de las transacciones (*Tid*); las tablas construidas se denominan *THT* (*Tid Hash Tables*, por sus siglas en inglés). El primero de los trabajos mencionados presenta el algoritmo *IHP* (*Inverted Hashing and Pruning*, por sus siglas en inglés). Este algoritmo construye un *THT* por cada ítem. Para comprender mejor, obsérvense las tablas 2 y 3, en las cuales se utiliza módulo 3 como función *hash*.

Tabla 2. Datos transaccionales y función *hash*

Hash	**TID**	**Transacción**	**Hash**	**TID**	**Transacción**
0	0	b	0	6	d, g
1	1	a, d, g	1	7	a, e, g
2	2	a, c, g	2	8	a, c, g
0	3	b, g	0	9	d, g
1	4	g	1	10	a, g
2	5	a, c	2	11	f, g

En la tabla 2 se tienen 12 transacciones y los ítems de las mismas pertenecen al conjunto $I = \{a, b, c, d, e, f, g\}$. Después de calcular el soporte de cada ítem se construye un *THT* por cada uno (tabla 3). Si se tiene un *minsup* igual a 3 ocurrencias, se pueden eliminar los *THT* asociados a los ítems no frecuentes b, e y f. En general, después del k-ésimo recorrido, $k \geq 1$, se pueden eliminar los *THT* de los ítems que no pertenezcan a ningún k-*itemset* frecuente, utilizándose los restantes *THT* para determinar los $(k+1)$-*itemsets* candidatos.

Tabla 3. *Tid Hash Table* de cada ítem

Hash	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0	2	0	2	0	0	3
1	3	0	0	1	1	0	4
2	3	0	3	0	0	1	3

Los *THT* son muy útiles para eliminar posibles conjuntos candidatos que no pueden ser frecuentes. Por ejemplo, aunque en la tabla 3 los 1-*itemsets* $\{a\}$ y $\{d\}$ son frecuentes, el conjunto {a, d} no puede ser candidato porque: al analizar los *THT* de $\{a\}$ y $\{d\}$ para cada valor *hash*, se concluye que el soporte del conjunto {a, d} es, a lo sumo, 1. Para ver detalles acerca de cuándo considerar un conjunto de ítems como candidato, referirse a [22].

En el segundo de los trabajos mencionados al principio de la sección se presenta el algoritmo *MIHP* (*Multipass with Inverted Hashing and Pruning*, por sus siglas en inglés)

[19]. El mismo es una combinación de los algoritmos *M-Apriori* (sección 3.1) e *IHP* (sección 3.1). La idea general del algoritmo es aprovechar las ventajas de *M-Apriori* en cuanto a reducir la memoria requerida mediante el particionamiento de los 1-*itemsets* frecuentes y el procesamiento de cada partición por separado; al mismo tiempo, utiliza el algoritmo *IHP* para podar, de forma eficiente, algunos de los posibles conjuntos candidatos.

Tanto *IHP* como *MIHP* muestran mejores resultados que los algoritmos anteriores a ellos, principalmente en conjuntos de datos con transacciones muy grandes (cientos de ítems por transacción). No obstante, cuando se tienen muchas transacciones, se reduce la efectividad de la poda debido a la alta densidad de transacciones por cada celda de la tabla *hash*. Además, cuando la cantidad de ítems de los conjuntos candidatos es grande y/o se tienen muchos conjuntos candidatos, resulta costoso verificar si un conjunto de ítems puede considerarse candidato o no.

FP-growth* y *Apriori-TFP

Los algoritmos presentados hasta el momento utilizan una estrategia de recorrido en amplitud y generan conjuntos candidatos de ítems. Seguidamente, aplican una heurística para podar algunos conjuntos de ítems que no pueden ser frecuentes y finalmente, calculan los soportes de los conjuntos de ítems que no fueron podados.

En esta sección se describen algunos de los algoritmos que siguen una estrategia de recorrido en profundidad. Estos algoritmos utilizan estructuras arbóreas para almacenar, de forma compacta, el conjunto de datos y calcular el soporte de los FI, además realizan sólo dos recorridos por el conjunto de datos.

El primero de los algoritmos descritos en esta sección es el *FP-growth* [15]. Paradigma de los algoritmos que usan estructuras arbóreas, se basa en el crecimiento o extensión de los FI y utiliza una estructura de datos compacta denominada *FP-tree*, también conocida como árbol de prefijos. Para describir la estructura *FP-tree* nos apoyaremos en la tabla 4 y la figura 6. En la primera se tiene un conjunto de transacciones y los 1-*itemsets* frecuentes, con *minsup* igual a 3 ocurrencias, que se obtienen después del primer recorrido.

Tabla 4. Conjunto de transacciones y $1-$ *itemsets* frecuentes

TID	**Transacción**	**1-itemsets Frecuentes**
0	f, a, c, d, g, i, m, p	f, c, a, m, p
1	a, b, c, f, l, m, o	f, c, a, b, m
2	b, f, h, j, o	f, b
3	b, c, k, s, p	c, b, p
4	a, f, c, e, l, p, m, n	f, c, a, m, p

Los 1-*itemsets* frecuentes se ordenan descendentemente de acuerdo a su soporte, esto permite una mayor compactación de la estructura de datos. La estructura de datos *FP-tree* es un árbol donde cada nodo se etiqueta con un ítem y su soporte. En el segundo recorrido cada transacción se inserta en el árbol, con los ítems ordenados descendentemente de acuerdo a su soporte. De esta forma, los prefijos iguales comparten la misma rama e incrementan en uno el valor de soporte de los ítems que forman la rama.

Además de la estructura de datos *FP-tree*, se crea una estructura llamada *HT* (*Header Table*) que almacena, para cada 1-*itemset* frecuente, su soporte y una referencia a su primera aparición en el *FP-tree*. Para hacer eficientes los recorridos en el *FP-tree* se enlazan los nodos con el mismo ítem.

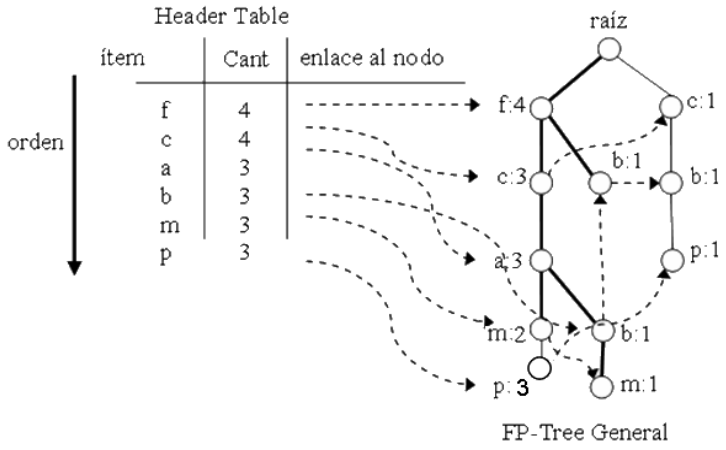

Fig. 6. Estructuras de datos *FP-tree* y *Header Table*

Después de construidas ambas estructuras se generan los FI, para ello se procesan los ítems uno a uno en orden ascendente del soporte. Se toma cada ítem i y se extrae el árbol cuyas ramas tienen al ítem i como nodo hoja; este nuevo árbol se procesa recursivamente de igual forma, pero sin considerar los nodos hojas. Cuando se obtiene un árbol de una sola rama se forman todos los conjuntos de ítems posibles con los ítems que forman la rama, y se les asigna el soporte del nodo hoja. Luego se regresa en retroceso extendiendo estos conjuntos con los ítems de las hojas eliminadas de los árboles anteriores y disminuyendo los soportes, en caso de ser menores.

Este algoritmo tiene la ventaja de compactar y almacenar el conjunto de datos en una estructura que contiene la información necesaria para realizar el cálculo del soporte de todos los conjuntos de ítems. *FP-growth*, al igual que el resto de los algoritmos que usan estructuras arbóreas y que serán presentados a continuación, son muy eficientes en conjuntos de datos densos[5] y poco dispersos, pero en conjuntos de datos muy dispersos con miles, incluso millones de ítems y transacciones, no se aprovecha la compactación que brinda el *FP-tree* resultando muy costosos los recorridos sobre el mismo.

El segundo algoritmo, presentado en [3], se nombra *Apriori-TFP* (*Apriori-Total from Partial*, por sus siglas en inglés) y muestra mejores resultados que el algoritmo *FP-growth*. Este algoritmo realiza un primer recorrido por el conjunto de datos y construye una estructura de datos denominada *P-tree* (*Partial tree*), en la que almacena los soportes parciales de todos los conjuntos de ítems. Seguidamente, realiza un segundo recorrido y a partir del *P-tree*, construye una segunda estructura de datos nombrada *T-tree* (*Total tree*). Después del segundo recorrido quedan todos los FI y sus soportes totales almacenados en la estructura *T-tree*. En los experimentos realizados en [3] se compara el algoritmo *Apriori-TFP* sólo con una implementación de *FP-growth* hecha por el autor en el lenguaje Java, la cual no necesariamente es óptima.

[5] Los conjuntos de datos densos, a diferencia de los dispersos, tienen un promedio de ítems por transacción muy alto, 30 % o más, con respecto al total de ítems.

PatriciaMine

Otro algoritmo muy referenciado, que muestra mejores resultados que *FP-growth*, es el algoritmo *PatriciaMine* [30]; este utiliza una estructura de datos llamada *PatriciaTrie* que compacta aún más el *FP-tree* descrito en la sección anterior. *PatriciaTrie* agrupa en un nodo todos los nodos consecutivos que tienen igual valor de soporte.

En la figura 7 se ilustran dos árboles, a la izquierda el *FP-tree* y a la derecha el mismo árbol pero compactado según *PatriciaTrie*.

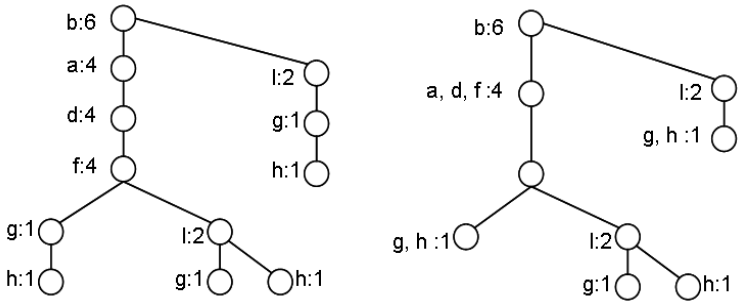

Fig. 7. Estructuras de datos *FP-tree* y *PatriciaTrie*

Además de usar una estructura más compacta, *PatriciaMine* realiza proyecciones del conjunto de datos sobre el conjunto de ítems, es decir, dado un conjunto de ítems X obtiene el conjunto de transacciones que contienen a X. La cantidad de transacciones obtenidas coincide con el soporte de X; este conjunto de transacciones se reutiliza eficientemente para obtener los soportes de los subconjuntos de X, aunque los autores no explican cómo. Este algoritmo, al igual que *FP-growth*, no se comporta de manera eficiente en conjuntos de datos muy dispersos, pues no puede explotar sus ventajas de compactación y los recorridos por la estructura son muy costosos.

CT-ITL y *CT-PRO*

El primero de estos algoritmos utiliza algunas de las ideas anteriores como son: los árboles de prefijos (*FP-tree*), un recorrido en profundidad extendiendo los FI y la intersección de listas de *TID* para el cálculo del soporte. El algoritmo *CT-ITL* [34] utiliza una estructura de datos denominada *CT-tree* (*Compress Transaction Tree*), que modifica la estructura FP-*tree* para almacenar grupos de transacciones.

La estructura *CT-tree* reduce la cantidad de nodos con respecto a las estructuras descritas anteriormente, permitiendo de esta forma un mejor agrupamiento de las transacciones que comparten un conjunto de ítems. Un árbol de prefijos completo contiene muchos sub-árboles idénticos, *CT-tree* almacena de forma unida la información contenida en los sub-árboles idénticos (Fig. 8). En la figura 8: *st1*, *st2* y *st3* son ejemplos de sub-árboles que se repiten en el árbol de prefijos completo.

Además de usar esta estructura de datos, *CT-ITL* se apoya en otra estructura llamada *ITL* (*Item Trans Link*, por sus siglas en inglés) para realizar una búsqueda eficiente de los FI.

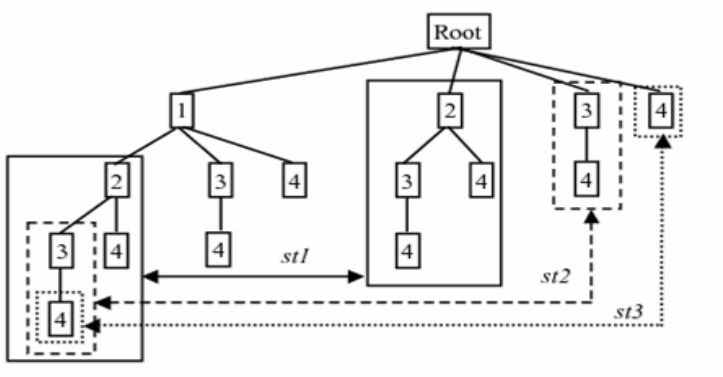

Fig. 8. Estructuras de datos *CT-tree*

El algoritmo *CT-PRO* [35] propone una estructura llamada *CFP-tree* (*Compressed FP-tree*), la cual puede reducir a la mitad el número de nodos del *FP-tree*. Para minar los FI, el algoritmo *CT-PRO* sigue una estrategia similar al algoritmo *CT-ITL*.

Todos los algoritmos que almacenan el conjunto de datos en estructuras arbóreas, presentan problemas con los conjuntos de datos muy grandes y dispersos. Cuando los datos son dispersos, las estructuras arbóreas resultan muy grandes debido a que compactan poco y los recorridos por ellas son muy costosos.

Eclat e *HybridMiner I*

Hasta el momento se han presentado algoritmos que siguen una estrategia de búsqueda descendente, es decir, realizan el recorrido desde los 1-*itemsets* hasta la frontera formada por los FI. En esta sección se presentan dos algoritmos que siguen una estrategia ascendente, aproximando los conjuntos de ítems potencialmente maximales[6] [39,23].

En [39] se presentan dos algoritmos para generar los conjuntos de ítems potencialmente maximales, uno basado en clases de equivalencia (ver sección 2.3) y otro basado en *hipergrafos-cliques*[7]. Ambos algoritmos realizan un balance entre la precisión con que estiman los conjuntos de ítems potencialmente maximales y el costo computacional. La aproximación por *hipergrafos-cliques* es más precisa pero tiene un alto costo computacional, mientras que la aproximación por clases de equivalencia sacrifica precisión pero tiene menor costo computacional. Por ser el algoritmo basado en clases de equivalencia más eficiente, se describe sólo éste, el cual se denomina *Eclat* (*Equivalence Class Transformation*, por sus siglas en inglés).

Según la definición de clase de equivalencia, los *k-itemsets* se pueden particionar, agrupándolos por prefijos iguales de tamaño $k-1$. El algoritmo *Eclat* desciende recursivamente por cada clase de equivalencia encontrada. El descenso se realiza con un procedimiento que extiende los ítems y calcula el soporte simultáneamente. Este procedimiento

[6] Un FI es maximal si no es subconjunto propio de ningún otro FI.

[7] Un *hipergrafo-clique*, denotado por $H(G)$, es un hipergrafo generado a partir del grafo G tal que:

- Vértices($H(G)$) = Vértices(G).
- Las hiperaristas de $H(G)$ forman cliques maximales en G.

opera sobre dos conjuntos de ítems X e Y, extendiéndolos mediante la operación $X \cup Y$. Por otra parte, en *Eclat* se considera una representación vertical del conjunto de datos y se determinan los soportes mediante la intersección de las listas de Tid asociadas con X y Y.

Aunque *Eclat* ha sido ampliamente referenciado, presenta como deficiencia la gran cantidad de conjuntos de ítems que procesa. No obstante, se han propuesto implementaciones eficientes de este método [4].

En [23] se presenta un algoritmo denominado *HybridMiner I*, que sigue una estrategia similar a *Eclat* pero reduce la cantidad de conjuntos de ítems procesados en casi un 50 %. En este mismo trabajo se presenta una segunda versión nombrada *HybridMiner II*, esta versión utiliza una estrategia de búsqueda descendente aproximando los conjuntos potencialmente minimales[8], pero sus resultados son menos eficientes que los de *HybridMiner I*. En los experimentos presentados en [23] se comparan, bajo las mismas condiciones, los algoritmos *HybridMiner I* y *Eclat*, siendo mejor el primero pero por diferencias muy pequeñas.

Al igual que *Eclat*, los algoritmos propuestos en este reporte particionan el espacio de búsqueda en clases de equivalencia, pero las utilizan de forma diferente, acumulando y reutilizando eficientemente el soporte de los prefijos de las mismas.

3.2. Algoritmos para datos dinámicos

En las secciones siguientes se presentan algunos algoritmos para minar todos los FI en conjuntos de datos dinámicos (que pueden cambiar), entiéndase por "algoritmo para conjunto de datos dinámico" un algoritmo que reutiliza el conjunto de FI calculados hasta el momento para calcular los nuevos FI después de cada actualización, sin tener que recorrer desde el principio todo el conjunto de datos.

La cantidad de trabajos relacionados para calcular los FI en datos dinámicos es mucho menor que en datos estáticos. En la siguiente sección se presentan dos algoritmos que siguen una estrategia de recorrido en amplitud. De los algoritmos para datos dinámicos que siguen esta estrategia, los dos presentados a continuación son los que tradicionalmente se refieren en la literatura, y ambos algoritmos son mucho más lentos que los algoritmos que siguen una estrategia de recorrido en profundidad.

De los algoritmos que siguen una estrategia en profundidad se describen cuatro basados en el algoritmo *FP-growth*, que modifican la estructura de datos *FP-tree*.

Otros algoritmos para datos dinámicos que no se explican en este trabajo tratan el problema de ítems faltantes [37], ítems asociados con una probabilidad existencial [12,26] y otros algoritmos tratan un área nueva denominada minería de flujo [6,7], la cual se diferencia del minado de datos dinámicos en que los flujos de datos cambian constantemente, por lo que no se puede recorrer dos veces el conjunto de datos, además de que la información se vuelve obsoleta después de cierta cantidad de transacciones.

Uno de los primeros algoritmos para calcular los FI en datos dinámicos se presenta en [9], el cual se denomina *FUP* (*Fast Update*, por sus siglas en inglés) y permite adicionar un conjunto de transacciones *db* a un conjunto de transacciones ya minado *DB*. El algoritmo *FUP*, para calcular los nuevos FI después de cada actualización, reutiliza los FI obtenidos

[8] Un conjunto de ítems es no frecuente minimal si no es superconjunto de ningún otro no frecuente.

en *DB*. Este algoritmo utiliza una estrategia de recorrido en amplitud y reduce en gran medida la cantidad de conjuntos candidatos.

Básicamente, la heurística del algoritmo *FUP* es la misma que la de los algoritmos *Apriori* y *DHP*. Las características principales que diferencian *FUP* de estos algoritmos son las siguientes:

- En la iteración k, el soporte de los k-*itemsets* frecuentes calculados en *DB* se actualiza con el conjunto de transacciones *db* para eliminar aquellos que dejen de ser frecuentes en $DB \cup db$, para lo cual sólo se necesita recorrer *db*.
- Al recorrer *db* se obtiene un conjunto de candidatos C_k formado por los k-*itemsets* que no son frecuentes en *DB*. El conjunto C_k se actualiza con el conjunto de transacciones *DB* para verificar si son frecuentes en $DB \cup db$.
- Los k-*itemsets* que pertenecen a C_k pueden ser eliminados si no son frecuentes en *db*. Esto se debe a que, por construcción del conjunto C_k, tampoco son frecuentes en *DB*, por lo que es imposible que lo sean en $DB \cup db$ (este algoritmo considera el soporte como un porcentaje del total de transacciones).
- El tamaño del nuevo conjunto de transacciones *db* disminuye con cada iteración debido a que se eliminan algunos ítems de las transacciones (similar a *DHP*).

Los experimentos realizados en [9] muestran que *FUP* es de 2 a 16 veces más rápido que ejecutar los algoritmos *Apriori* o *DHP*, sobre todo el conjunto de datos.

En [10] se describe una variante de *FUP* llamada *FUP2*, la cual incluye la eliminación y la modificación de transacciones. En el caso de la adición de transacciones, *FUP2* es equivalente a *FUP*. EL algoritmo *FUP2* sólo es eficiente cuando se realizan pequeñas actualizaciones en el conjunto de datos. Ambos algoritmos presentan los mismos problemas que *Apriori* y todos los algoritmos que realizan generación de candidatos; todos realizan varios recorridos por el conjunto de datos y ante umbrales de soportes muy bajos generan conjuntos de candidatos muy grandes.

Los algoritmos que se describen en esta sección modifican la estructura de datos *FP-tree* vista en 3.1 para recalcular los FI después de cada actualización sobre el conjunto de datos.

En [11] se propone un algoritmo que utiliza una estructura de datos denominada *CATS Tree* (*Compressed and Arranged Transaction Sequences Tree*, por sus siglas en inglés), la cual extiende la idea de *FP-tree* para lograr una mejor compresión de los datos almacenados. El algoritmo propuesto se denomina FELINE (FrEquent/Large patterns mINing with CATS trEe), éste realiza un solo recorrido por el conjunto de datos y permite calcular los FI con diferentes soportes sin tener que reconstruir el árbol. Además, permite insertar y eliminar transacciones en todo momento, pero no utiliza los FI previamente minados para calcular los nuevos FI después de una actualización, sino que modifica el árbol y recalcula todos los FI como si fuera la primera vez.

El algoritmo *FELINE* supone que los datos siempre cabrán en memoria. Por tanto, la inserción o eliminación de una transacción en el árbol es eficiente, pero el tiempo de ejecución necesario para calcular los FI aumenta al aumentar el tamaño del árbol; los mejores resultados se obtienen cuando se mantiene el conjunto de datos invariable y se modifica el soporte. Aunque este algoritmo realiza un solo recorrido por el conjunto de

datos, la construcción del árbol es muy compleja. Al adicionar una nueva transacción en la estructura *CATS Tree*, se busca la rama del árbol que tenga más ítems en común con ella y se mezclan, trayendo como consecuencia altos tiempos de ejecución. Por otro lado, las modificaciones del conjunto de datos pueden implicar la necesidad de realizar operaciones *merging*, *swapping* y *splitting* sobre los nodos del árbol.

En [24] se presenta un algoritmo nombrado *AFPIM* (*Adjusting FP-tree for Incremental Mining*, por sus siglas en inglés) el cual ajusta la estructura de datos *FP-tree* para actualizar los FI después de la adición o eliminación de transacciones. *AFPIM* realiza dos recorridos por el conjunto de datos y al igual que en el algoritmo *FELINE*, las actualizaciones del conjunto de datos pueden implicar la necesidad de operaciones *merging*, *swapping* y *splitting* sobre los nodos del árbol. La adición o eliminación de transacciones puede cambiar el orden de los nodos en el árbol, de ahí la necesidad de las operaciones anteriores.

Las deficiencias de *FELINE* y *AFPIM* son resueltas en [27], en este trabajo se propone un algoritmo que utiliza una estructura de datos denominada *Cantree* (*Canonical-order Tree*, por sus siglas en inglés). Este algoritmo realiza un solo recorrido por el conjunto de datos y almacena la información en un árbol de prefijos similar al *FP-tree*; usa como técnica de minado el *FP-growth* y permite adicionar y eliminar transacciones (al igual que [11] y [24], supone que el árbol cabe en memoria). Como una ventaja de este algoritmo se tiene que no necesita realizar ajustes sobre el árbol ni realizar operaciones costosas sobre los nodos del árbol, como las mencionadas anteriormente. Aunque este algoritmo muestra mejores resultados que los dos anteriores, tiene la gran desventaja de que no ordena los ítems por su valor de soporte, lo cual trae como consecuencia que la estructura *Cantree* compacte menos que el *FP-tree* y el costo de calcular los FI, en conjuntos de datos muy dispersos, es muy alto.

En [36] se presenta una estructura de datos llamada *CP-tree* (*Compact Pattern tree*, por sus siglas en inglés), la cual almacena la información necesaria para calcular los FI realizando un solo recorrido por el conjunto de datos (*insertion phase*) y mantiene el mismo desempeño del algoritmo *FP-growth* mediante una reestructuración dinámica del árbol (*restructuring phase*). Los experimentos realizados por los autores muestran mejores resultados que los obtenidos en [27] utilizando la estructura *Cantree*. Tanto este algoritmo como todos los expuestos en esta sección presentan el problema de suponer que el conjunto de datos cabe en la memoria; además, al procesar conjuntos de datos muy grandes y dispersos, las estructuras de datos se hacen muy grandes y resulta costoso recorrerlas para actualizar los FI.

El algoritmo para datos dinámicos presentado en este reporte (*ICA*) no supone que el conjunto de datos cabe en la memoria, para almacenar los FI previamente calculados utiliza ficheros binarios y reutiliza estos FI para calcular los nuevos FI.

4. Algoritmo *CA*

Los algoritmos existentes para minar todos los FI en conjuntos de datos estáticos presentan dificultades al minar conjuntos de datos muy grandes y dispersos, díganse conjuntos de datos de millones de transacciones y miles o millones de ítems diferentes.

Los algoritmos que siguen una estrategia en amplitud necesitan mantener en memoria todos los *k-itemsets* frecuentes para generar los $(k+1)$-*itemsets* candidatos. Cuando se tienen muchos ítems diferentes o el umbral de soporte es muy bajo, la cantidad de conjuntos candidatos es muy grande; además, estos algoritmos realizan muchos recorridos por el conjunto de datos. En el caso de los algoritmos que siguen una estrategia en profundidad, se necesita mantener todo el conjunto de datos en memoria. Las estructuras de datos arbóreas compactan el conjunto de datos, almacenando en una misma rama las transacciones con ítems comunes. No obstante, cuando se tienen conjuntos de datos muy dispersos no existen muchas coincidencias entre las transacciones, lo cual resulta, en árboles muy grandes y en recorridos sobre éstos, muy costosos.

En esta sección se presenta un algoritmo para minar todos los FI en conjuntos de datos grandes y dispersos, el cual se denomina *CA* (*Compressed Arrays*, por sus siglas en inglés).

4.1. Algoritmo *CA*

A continuación se presenta el algoritmo *CA* así como su pseudocódigo. *CA* sigue una estrategia de recorrido en amplitud por cada clase de equivalencia (sección 2.3). La eficiencia del algoritmo *CA* se debe principalmente a la forma en que aprovecha la manera de almacenar la matriz de datos (*VVTid*). Como se verá más adelante, el almacenamiento vertical de esta matriz, combinado con la estructuración del espacio de búsqueda en clases de equivalencia, logra un menor consumo de memoria y un rápido cálculo del soporte de los FI.

De forma general, el algoritmo *CA* almacena la presencia de un ítem en una transacción con un bit igual a 1 y la ausencia con un bit igual a 0, logrando almacenar en un entero la presencia o ausencia de un ítem en 32 ó 64 transacciones, según la arquitectura de la computadora donde se ejecute el algoritmo. En la figura 9 se muestra un ejemplo para el caso de una arquitectura de 32 bits. En la sección **a)** de la figura 9 se muestra un conjunto de transacciones. En la sección **b)** se representa la presencia de un ítem en una transacción con un 1 y la ausencia con un 0. Finalmente, en la sección **c)**, se toman las secuencias de 32 valores consecutivos de 1s ó 0s, verticalmente por cada ítem, y se convierten en el número entero correspondiente a la secuencia de *bits* con esos valores. Por ejemplo, las primeras 32 transacciones del ítem 1 se convierten en el entero b_{11}, las siguientes 32 transacciones se convierten en el entero b_{12}, etc.

Después de almacenarse el conjunto de datos en la forma descrita, se crean las clases de equivalencia de tamaño 2 y para cada una de éstas se sigue una estrategia de recorrido en amplitud hasta obtener todos los FI derivados de la misma. El algoritmo *CA* aprovecha que todos los conjuntos de una clase de equivalencia comparten un prefijo común y mantiene, por cada clase de equivalencia, un arreglo de enteros con el soporte acumulado de los ítems que forman el prefijo. Cada entero de este arreglo representa 32 ó 64 transacciones según la arquitectura y sólo se almacenan los enteros diferentes de 0, ya que los iguales a 0 no aportan nada al cálculo del soporte. Los algoritmos anteriores que almacenan los datos en el formato *VVTid* [33,5,28] mantienen un arreglo de enteros para almacenar el soporte de cada FI, ocupando más memoria que el algoritmo CA. Además, en [33,5,28] se almacenan los enteros iguales a 0 realizándose operaciones de intersección innecesarias.

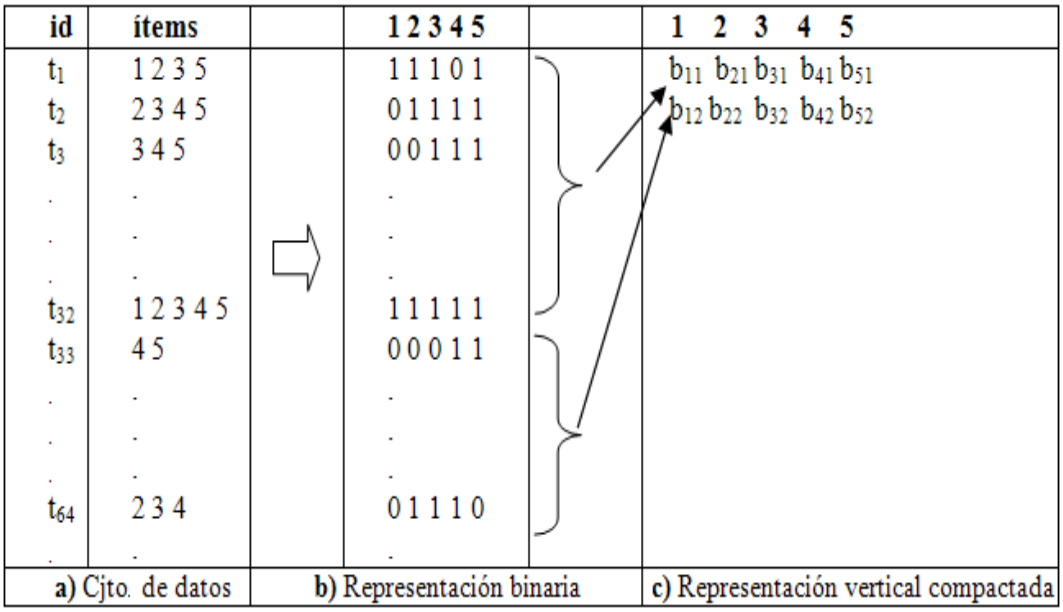

Fig. 9. Ejemplo del almacenamiento de un conjunto de datos en arquitecturas de 32 bits

Para calcular el soporte de un conjunto de ítems perteneciente a una clase de equivalencia, sólo hay que intersectar el arreglo de enteros que almacena el soporte del prefijo con el arreglo de enteros asociado al ítem sufijo.

Formalizando la descripción anterior del algoritmo, sea M la representación binaria de un conjunto de datos con n ítems y m transacciones, sea además w la longitud de palabra del *CPU*. Tomando de M las columnas asociadas a los ítems frecuentes, se representa el vector binario asociado a cada ítem j como un arreglo de enteros I_j, donde cada entero tiene longitud w, como sigue:

$$I_j = \{W_{1,j}, W_{2,j}, \ldots, W_{q,j}\},\ q = \lceil m/w \rceil \tag{5}$$

cada entero del arreglo I_j se define como:

$$W_{k,j} = \sum_{r=1}^{w} 2^{w-r} * M_{((k-1)*w+r),j} \tag{6}$$

siendo $M_{i,j}$ el valor de bit del ítem j en la transacción i, en el caso de ser $i > m$ se define $M_{i,j} = 0$.

Al estructurar el espacio de búsqueda en clases de equivalencia, se obtiene para cada 1-*itemset* i una clase de equivalencia cuyo prefijo es i y cuyos sufijos son todos los ítems lexicográficamente mayores que i. Este conjunto de clases de equivalencia agrupa a todos los conjuntos candidatos de tamaño 2. Como se puede observar en la figura 10, cada clase de equivalencia del nivel k genera un conjunto de clases de equivalencia en el nivel $k+1$. Además, en el nivel k-ésimo, el conjunto de clases de equivalencia que se obtiene forma una partición del conjunto de k-*itemsets*.

Como se vio en la sección 2.3, el retículo formado por los conjuntos de ítems se puede particionar en árboles donde cada 1-*itemset* es raíz de un árbol. El árbol asociado al primer

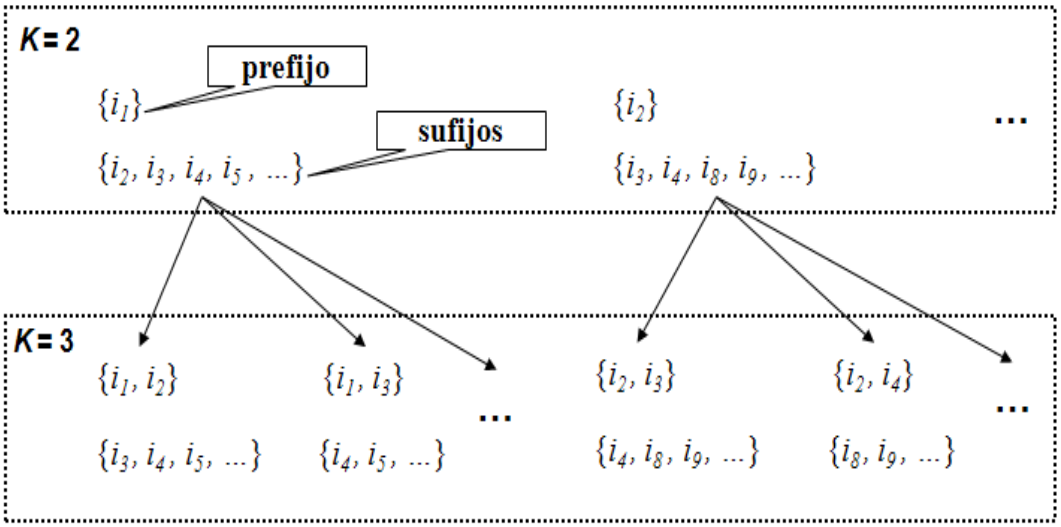

Fig. 10. Clases de equivalencia de los 2-*itemsets* y los 3-*itemsets*

ítem es la mitad del espacio de búsqueda, el árbol asociado al segundo es la cuarta parte del espacio de búsqueda y así sucesivamente; esto implica que, si se toma como prefijo de la primera clase de equivalencia de tamaño dos[9], un ítem i con soporte muy alto, la cantidad de 2-*itemsets* frecuentes que contengan a i será muy grande y por consiguiente, la cantidad de clases de equivalencia generadas en los próximos niveles también será grande.

Para eliminar ese problema, el algoritmo *CA* ordena los 1-*itemsets* frecuentes en orden ascendente de su soporte, de esta forma cada árbol minimiza su altura y el algoritmo procesa más rápido cada clase de equivalencia.

De forma general, *CA* genera iterativamente una lista de clases de equivalencia EC_k, los elementos de esta lista representan las clases de equivalencia de longitud k y tienen el formato:

$$\langle Prefix_{k-1}, IA_{Prefix_{k-1}}, Suffixes_{Prefix_{k-1}} \rangle, \tag{7}$$

donde $Prefix_{k-1}$ es el prefijo de longitud $(k-1)$ común a todos los conjuntos de ítems agrupados por la clase, $Suffixes_{Prefix_{k-1}}$ es el conjunto de los ítems j que se extienden con $Prefix_{k-1}$, donde j es lexicográficamente mayor que cada ítem de $Prefix_{k-1}$, e $IA_{Prefix_{k-1}}$, es un arreglo de enteros no nulos que almacena el soporte de los ítems que pertenecen a $Prefix_{k-1}$. Para ello realiza operaciones *AND* sobre los enteros de los arreglos asociados a cada ítem del prefijo $Prefix_{k-1}$. A medida que los conjuntos de ítems son más grandes, el arreglo IA tendrá menos elementos, esto se debe a que la operación *AND* genera muchos ceros, los cuales no es necesario almacenar, logrando así un menor consumo de memoria y un conteo de soporte más rápido. Sean i y j dos ítems frecuentes, IA se obtiene de la siguiente forma:

$$IA_{\{i\}\cup\{j\}} = \{(W_{k,i} \,\&\, W_{k,j}, k) \mid (W_{k,i} \,\&\, W_{k,j}) \neq 0, k \in [1, q]\} \tag{8}$$

[9] El tamaño de una clase de equivalencia está dado por la cardinalidad de los conjuntos de ítems que ésta agrupa (sección 2.3)

de igual forma, sean el conjunto de ítems frecuentes X y el ítem frecuente j

$$IA_{X\cup\{j\}} = \{(b \,\&\, W_{k,j}, k) \mid (b,k) \in IA_X, b \,\&\, W_{k,j} \neq 0, k \in [1,q]\}. \tag{9}$$

El pseudocódigo de *CA* se muestra en el algoritmo 4.1.

Algoritmo 4.1: CA

Input: Conjunto de datos almacenados en *VTTid*
Output: Todos los FI

```
1  Answer = ∅
2  L = {1-itemsets frecuentes}
3  forall i ∈ L do
4      ECGenAndCount(⟨{i}, I_i, Suffixes_{i}⟩, EC_2)
5      k = 3
6      while EC_{k-1} ≠ ∅ do
7          forall ec ∈ EC_{k-1} do
8              ECGenAndCount(ec, EC_k)
9          end
10         Answer = Answer ∪ EC_k
11         k = k + 1
12     end
13 end
14 return Answer
```

El pseudocódigo del algorimo *CA* es similar al del algoritmo *APriori*, pero aplicado a cada clase de equivalencia. En las líneas 4 y 8 se invoca la función *ECGenAndCount*, la cual toma como argumento una clase de equivalencia de tamaño $k-1$ y genera todas las clases de equivalencia de tamaño k derivadas de ésta (Algoritmo 4.2).

En la línea 2 del pseudocódigo del algoritmo 4.2 se recorren todos los ítems i que forman el sufijo de la clase de equivalencia de entrada (EC_{k-1}). En la línea 3 se construyen los prefijos $Prefix'$ de las clases de equivalencia del nivel k adicionando cada uno de los sufijos i al prefijo de EC_{k-1}. En la línea 4 se calcula el arreglo IA asociado a cada $Prefix'$ mediante una operación de *AND* entre el IA de EC_{k-1} y el I_i asociado al ítem i (expresión 9). Es importante resaltar que para realizar la intersección se recorre IA, el cual sólo almacena los bloques de 32 transacciones diferentes de 0, resultando más eficiente la intersección. De la línea 6 a la 8 se recorren los sufijos j de EC_{k-1} que son lexicográficamente mayores que i y se verifica el soporte de los conjuntos $Prefix' \cup j$ realizando una operación de *AND* entre el IA de $Prefix'$ y el I_j asociado al ítem j.

En la línea 7 se utiliza la función *Support* para calcular el soporte de cada conjunto de ítems. Esta función, dado un conjunto de ítems X, el cual tiene a IA_X como intersección de los vectores binarios asociados a los ítems que lo forman, calcula la cantidad de bits iguales a 1 en el arreglo de enteros IA_X.

4.2. Consideraciones de memoria

Como se describió en 2.1, se han reportado en la literatura cuatro formas de almacenar la matriz de datos, dos horizontales (*LHI* y *VHI*) y dos verticales (*LVTid* y *VVTid*). Todos

Algoritmo 4.2: ECGenAndCount

Input: Una clase de equivalencia con formato $\langle Prefix, IA_{Prefix}, Suffixes_{Prefix}\rangle$
Output: Un conjunto de clases de equivalencia

```
1  Answer = ∅
2  forall i ∈ Suffixes_Prefix do
3      Prefix' = Prefix ∪ {i}
4      IA_Prefix' = IA_Prefix ∪ {i}
5      Suffixes'_Prefix' = ∅
6      forall j ∈ Suffixes_Prefix , j lexicográficamente mayor que i do
7          if Support(IA_Prefix' ∪ {j}) then
8              Suffixes'_Prefix' = Suffixes'_Prefix' ∪ {j}
9          end
10     end
11     if Suffixes'_Prefix' ≠ ∅ then
12         Answer = Answer ∪ {⟨Prefix', IA_Prefix', Suffixes'_Prefix'⟩}
13     end
14 end
15 return Answer
```

los autores coinciden en las ventajas del almacenamiento vertical sobre el horizontal, esta ventaja se debe principalmente a que el almacenamiento vertical permite calcular los soportes de los conjuntos de ítems mediante intersecciones de listas o arreglos de enteros, según sea el caso. Tomar una decisión, referente al consumo de memoria, entre las representaciones *LVTid* y *VVTid* no es una tarea trivial. En [5], los autores analizaron estos dos formatos y concluyeron experimentalmente, que la eficiencia en memoria depende de la densidad de los datos; particularmente, en arquitecturas de 32 bits el formato *LVTid* resulta más costoso en términos de memoria si el soporte de los conjuntos de ítems es mayor que 1/32 o cercano al 3 %. En este formato, se necesita un entero para representar la presencia de un ítem contra un simple bit en el formato *VVTid*.

En el algoritmo *CA* se necesita, por cada clase de equivalencia, un par de enteros por cada 32 transacciones, uno para el bloque de 32 transacciones y el otro para el identificador del bloque (sólo se almacenan en el arreglo IA los enteros no nulos). La sobrecarga de memoria de este formato es mayor que la del formato *VVTid* si la cantidad de bloques diferentes de 0 es mayor que el 50 % del total de bloques.

Considerando un conjunto de datos con m transacciones en una arquitectura de 32 bits y un umbral de soporte igual a *minsup*, el formato *VVTid* requiere $m/8$ bytes de memoria mientras que *CA*, en el peor de los casos, cuando el conjunto de datos es disperso requiere $8 * \text{mín}\{m * minsup, m/32\}$ bytes. A medida que el umbral de soporte *minsup* sea más pequeño, más eficiente en memoria será el algoritmo *CA*.

4.3. Experimentación y resultados

Para evaluar el desempeño del algoritmo *CA* se realizaron dos experimentos diferentes. En el primer experimento se comparó el algoritmo *CA* con los algoritmos más eficientes descritos en la sección 3.1, no se mostraron las comparaciones con el algoritmo *Apriori* por ser notablemente mayor el tiempo de ejecución del mismo comparado con el del resto de

los algoritmos. Las implementaciones seleccionadas para experimentar fueron: *Fp-growth* y *Eclat* obtenidas de [Bodon-url] y el algoritmo *PatriciaMine* obtenido de [30]. En el segundo experimento se evaluó el escalamiento[10] de *CA* con respecto a los algoritmos *Fp-growth* y *Eclat*. Para ello se tomó un conjunto de datos de aproximadamente un millón de transacciones y se construyeron 10 conjuntos de datos formados por las primeras cien mil transacciones, las primeras doscientas mil transacciones y así sucesivamente hasta el conjunto de datos completo. En este experimento no se tuvo en cuenta el algoritmo *PatriciaMine* por no tener buenos resultados en los conjuntos de datos grandes y dispersos.

El primer experimento fue desarrollado con tres conjuntos de datos de noticias (*El Pais*, *TDT* y *Kosarak*), una colección de documentos web (*Webdocs*) y dos conjuntos de datos sintéticos (*Chess* y *T40I10D100K*). Algunos de estos conjuntos de datos son muy densos, como es el caso de *Chess*, otros son dispersos como *El País* y otros muy dispersos como *Webdocs* (Tabla 5). Los conjuntos de datos de noticias fueron lematizados[11] usando la aplicación *Treetagger* [32] y las *stopwords* fueron eliminadas.

Tabla 5. Características de los conjuntos de datos

	Cant. de Trans.	Cant. de Ítems	Prom. de Ítems / Trans.
Chess	3196	75	37
T40I10D100K	100000	942	39.54
El Pais	550	14489	173.1
TDT	8169	55532	133.5
Kosarak	990002	41935	8.1
Webdocs	1704140	5266562	175.98

El segundo experimento se realizó con el conjunto de datos *Kosarak*. El conjunto de datos *Kosarak* fue donado por Ferenc Bodon al repositorio FIMI [FIMI-url] y contiene datos de un portal de noticias húngaro. El conjunto de datos *Webdocs* contiene una colección de documentos *web* y fue donada al repositorio FIMI por Claudio Lucchese *et al.* El conjunto de datos *TDT* contiene noticias colectadas diariamente de 6 fuentes noticiosas en *American English* por un período de seis meses (Enero - Junio, 1998). El conjunto de datos *El País* contiene 550 noticias, publicadas en diario con igual nombre durante junio de 1999. El resto de los conjuntos de datos (*Chess* y *T40I10D100K*) fue descargado del repositorio *FIMI*; estos conjuntos fueron generados con un generador que recibe varios parámetros como: cantidad de ítems diferentes, por ciento de ítems por transacción, cantidad de transacciones, entre otros.

Los experimentos fueron realizados en una PC con un Intel Core 2 Duo a 1.86 GHz CPU y 1 GB DDR2 RAM, se usó Windows XP SP2 y todos los algoritmos se implementaron en el lenguaje de programación ANSI C. En el tiempo final de ejecución se consideraron los tiempos de entrada y salida (CPU+IO) para todos los algoritmos.

Como se puede ver en los gráficos del 11 al 16, el algoritmo *CA* es más rápido que el resto de los algoritmos en la mayoría de los conjuntos de datos, aunque es superado por el algoritmo *PatriciaMine* en los conjuntos de datos *T40I10D100K* y *El País* para los

[10] El escalamiento de un algoritmo se refiere a la forma en que varía una medida en general respecto a otra, en nuestro caso se refiere a cómo varía el tiempo respecto a la cantidad de transacciones.

[11] La técnica de lematización consiste en la reducción de variantes morfológicas de la formas de una palabra a raíces comunes o lexemas.

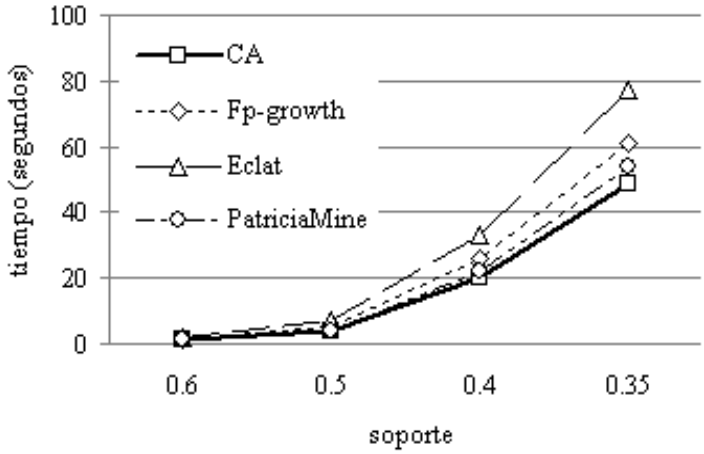

Fig. 11. Comparativa de tiempos de ejecución (*Chess*)

valores de soporte más pequeños. El caso particular del algoritmo *PatriciaMine* muestra muy buenos resultados en los conjuntos de datos más pequeños, aunque en los grandes y dispersos, el tiempo de ejecución tarda más de una hora. En éstos casos los resultados del algoritmo no fueron graficados (Figuras 15 y 16).

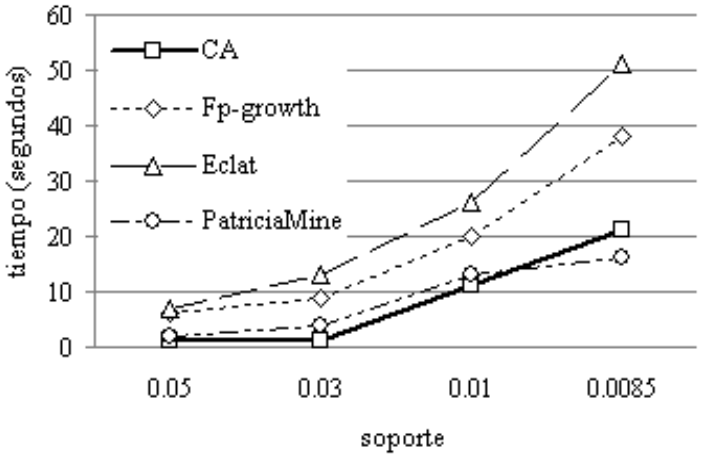

Fig. 12. Comparativa de tiempos de ejecución (*T40I10D100K*)

La mayor diferencia del algoritmo *CA* respecto al resto de los algoritmos se obtiene en los conjuntos de datos *Kosarak* y *Webdocs* (Figuras 15 y 16). Al ser estos conjuntos de datos muy grandes y dispersos, las estructuras arbóreas construidas por los otros algoritmos para almacenar los datos, tienen una cantidad de ramas casi igual a la cantidad de transacciones del conjunto de datos, implicando recorridos muy costosos en tiempo. Se puede notar cómo al disminuir el soporte, aumentan los tiempos de ejecución; esto se debe a que aumenta la cantidad de FI generados.

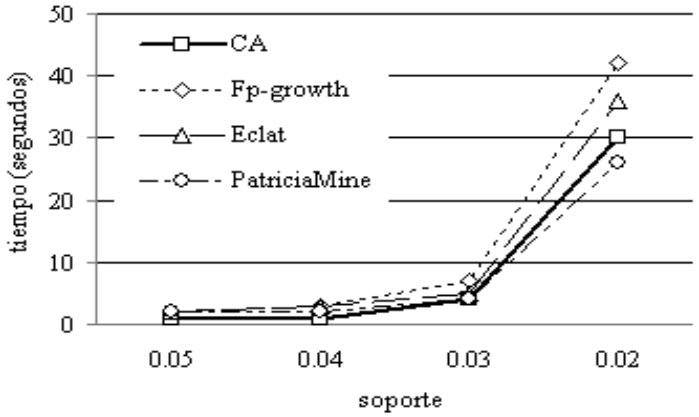

Fig. 13. Comparativa de tiempos de ejecución (*El País*)

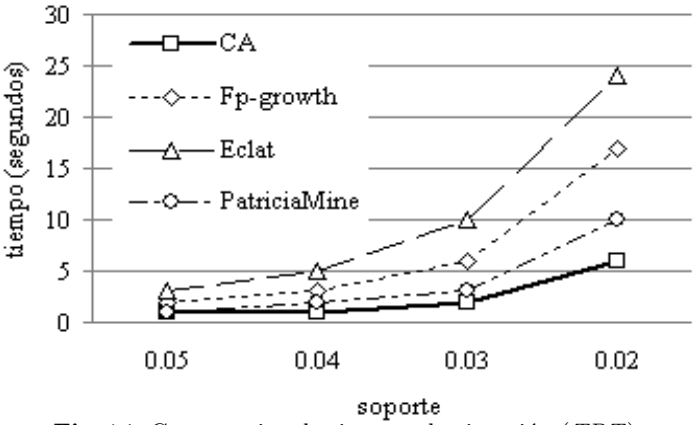

Fig. 14. Comparativa de tiempos de ejecución (*TDT*)

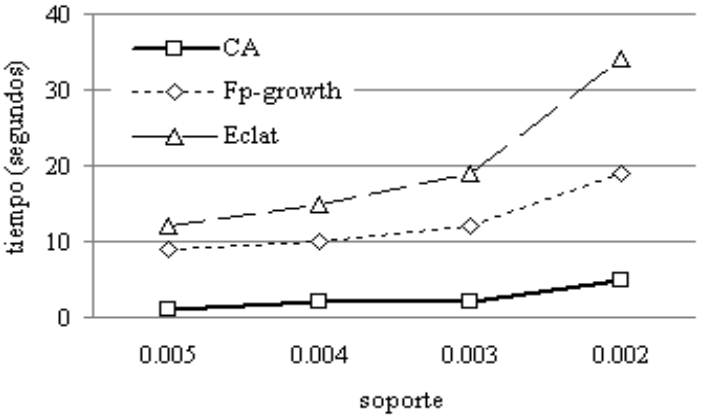

Fig. 15. Comparativa de tiempos de ejecución (*Kosarak*)

En la figura 17 se observa que en los tres algoritmos el tiempo de ejecución crece linealmente con respecto a la cantidad de transacciones. En *Fp-growth* y *Eclat*, las diferencias de tiempo, al adicionar cien mil transacciones, varían entre 3 y 4 segundos, mientras que en el caso del algoritmo *CA*, varían entre 1 y 2 segundos. Por tanto, *CA* escala con menor pendiente logrando los mejores resultados. En la sección 4 se puede ver un experimento similar con tres de los conjuntos de datos restantes obteniéndose el mismo resultado.

Los experimentos muestran que el algoritmo *CA* tiene un buen desempeño tanto en conjuntos de datos densos como dispersos. No obstante, en conjuntos de datos pequeños cuyas transacciones compartan muchos ítems en común, algoritmos como el *PatriciaMine*, que aprovecha esta característica, obtienen mejores resultados. *CA* es más conveniente para conjuntos de datos grandes y dispersos ya que los arreglos de enteros asociados a las clases de equivalencia reducen su cardinalidad rápidamente favoreciendo el cálculo del soporte.

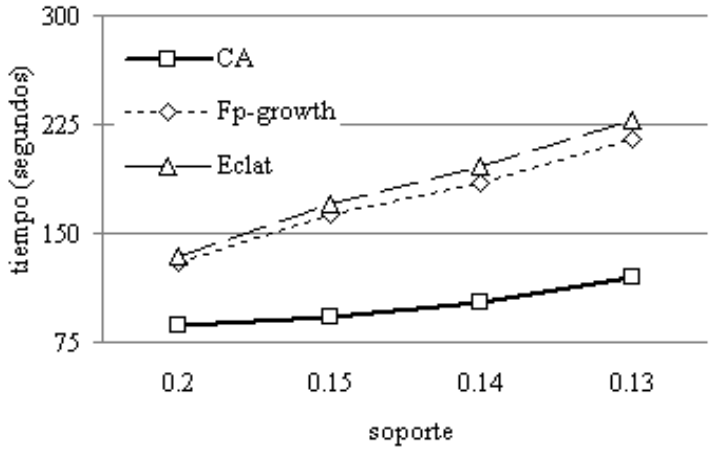

Fig. 16. Comparativa de tiempos de ejecución (*Webdocs*).

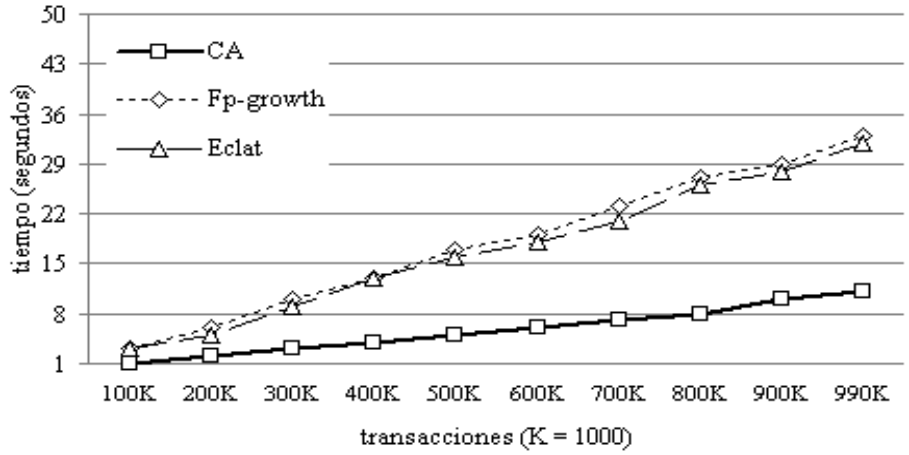

Fig. 17. Escalamiento con soporte 0,002 (*Kosarak*).

5. Algoritmo *ICA*

En la sección anterior se describió un algoritmo denominado *CA* para calcular todos los FI en conjuntos de datos estáticos. Muchos de los sistemas desarrollados hoy en día actualizan constantemente sus datos ya sea adicionando, eliminando o modificando información. Por muy eficiente que sea un algoritmo para calcular los FI en conjuntos de datos estáticos, si los datos cambian, tendrá que procesar todo el conjunto de datos desde el inicio para recalcular los FI.

En esta sección se describe un algoritmo para minar todos los FI en conjuntos de datos dinámicos, el cual se denomina *ICA* (*Incremental Compressed Arrays*) y se basa en el algoritmo *CA* previamente descrito.

5.1. Algoritmo *ICA*

A continuación se presenta el algoritmo *ICA* así como su pseudocódigo. *ICA* permite adicionar, eliminar y modificar transacciones con soporte variable e invariable. A diferencia de la mayoría de los algoritmos existentes para el minado de todos los FI en conjuntos de datos dinámicos, el algoritmo *ICA* no usa ninguna estructura arbórea para almacenar los FI previamente calculados. Para almacenar los FI previamente calculados, el algoritmo *ICA* crea un registro binario[12] por cada nivel k (longitud de los conjuntos de ítems analizados).

ICA mantiene la misma representación del conjunto de datos y la misma estructuración en clases de equivalencia que el algoritmo *CA*. La principal diferencia entre ambos algoritmos es que *CA* necesita procesar todo el conjunto de datos después de cada actualización, mientras que *ICA* almacena los FI actuales de forma eficiente y los reutiliza para calcular los FI después de cada actualización (adición, eliminación o modificación).

Además de los archivos binarios con los FI de cada nivel, se mantiene un archivo binario con todos los ítems diferentes y sus soportes. Es importante resaltar que el orden lexicográfico de los ítems no cambia aunque el ítem se vuelva no frecuente después de una actualización.

Dentro de cada archivo binario los FI se almacenan ordenados por clases de equivalencia. Como en todo momento existe un orden lexicográfico para los ítems, este orden define un orden lexicográfico entre las clases de equivalencia. Para cada clase de equivalencia se almacena primero el prefijo y después cada sufijo seguido por su soporte.

En la figura 18, se muestra una clase de equivalencia de prefijo $\{i_1, i_2\}$, sufijos $\{i_3, i_4, i_5, i_6\}$ y soportes $\{6, 12, 4, 8\}$. El soporte del conjunto $\{i_1, i_2, i_3\}$ es 6, el soporte de $\{i_1, i_2, i_4\}$ es 12 y así sucesivamente. Debajo de la clase de equivalencia se muestra el orden en que se almacena la misma en el archivo binario.

Cuando el algoritmo *ICA* carga el archivo binario de un nivel para actualizar los soportes de algunos FI o para adicionar o eliminar FIs, necesita realizar una búsqueda rápida. Si al almacenar la clase de equivalencia se almacenaran los valores de los ítems, la búsqueda sería lineal porque los valores de los ítems no tienen un orden. Por ello, en vez de almacenar el valor del ítem en el archivo binario se almacena su valor lexicográfico, el cual no cambia nunca. De esta forma, *ICA* realiza la búsqueda de un FI en tiempo $O(\log n)$, la adición de

[12] Los registros binarios almacenan un flujo de 0s y 1s.

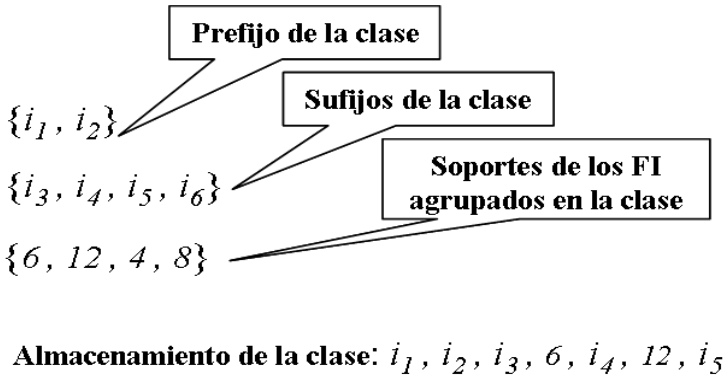

Fig. 18. Orden en que se almacena una clase de equivalencia en un registro.

un FI en tiempo $O(\log n)$ y la adición de una clase de equivalencia en tiempo $O(n)$, siendo n la cantidad de clases de equivalencia del nivel actual.

En el algoritmo 5.1 se muestra el pseudocódigo del algoritmo *ICA*:

Algoritmo 5.1: ICA

Input: Conjunto de datos almacenados en *VTTid*
Output: Todos los FI

```
1  EC_1 = {1-itemsets frecuentes} //1-itemsets frecuentes globales
2  k = 2
3  while EC_{k-1} ≠ ∅ do
4      EC_k = ∅,   KFreq = ∅
5      LoadKFrequents(KFreq)
6      forall ec ∈ EC_{k-1} do
7          ECGenAndCountDin(ec, KFreq, EC_k)
8      end
9      WriteUpdatedKFrequents(KFreq) //KFreq se actualiza con EC_k
10     k = k + 1
11 end
```

En la línea 1 del pseudocódigo del algoritmo 5.1 se calculan los 1-itemsets frecuentes globales, para ello se utiliza el archivo que almacena todos los ítems diferentes y sus soportes. Es importante aclarar que sólo los ítems que sean frecuentes globales pueden generar nuevos FI.

El algoritmo funciona de forma similar al algoritmo *CA* realizando un recorrido en amplitud por cada clase de equivalencia. En cada nivel k, *ICA* carga el archivo binario que almacena los FI actuales de tamaño k (línea 5). En la línea 7 se invoca la función *ECGenAndCountDin* (Algoritmo 5.2), la cual toma como argumento un conjunto de $(k-1)$-itemsets frecuentes (EC_{k-1}), el conjunto de k-itemsets frecuentes previamente calculado ($KFreq$) y actualiza todas las clases de equivalencia EC_k cuyos elementos son k-itemsets frecuentes.

La función *WriteUpdatedKFrequents* (línea 8) actualiza en el archivo binario correspondiente los FI e ignora los conjuntos de ítems que se volvieron no frecuentes debido a un aumento del soporte o a la actualización de los datos.

Algoritmo 5.2: ECGenAndCountDin

Input: Una clase de equivalencia con formato $\langle Prefix, IA_{Prefix}, Suffixes_{Prefix}\rangle$ (EC_{k-1}), los k-itemsets frecuentes actuales ($KFreq$) y un conjunto de clases de equivalencia EC_k (inicialmente vacío)

Output: El conjunto de clases de equivalencia generado (EC_k actualizado)

```
1  Answer = EC_k
2  forall i ∈ Suffixes_Prefix do
3      Prefix' = Prefix ∪ {i}
4      IA_Prefix' = IA_Prefix ∪ {i}
5      Suffixes'_Prefix' = ∅
6      forall j ∈ Suffixes_Prefix , j lexicográficamente mayor que i do
7          Sup = Support(IA_Prefix' ∪ {j})
8          if (Prefix' ∪ {j} ∈ KFreq) y (sea Sup' su anterior soporte) then
9              Sup = Sup + Sup'    //en caso de eliminación Sup = −Sup + Sup'
10             Sup' = Sup    //el soporte de Prefix' ∪ {j} se actualiza en KFreq
11         end
12         else
13             Sup = Sup + Rerun(Prefix' ∪ {j})
14         end
15         if Sup > minSup then
16             Suffixes'_Prefix' = Suffixes'_Prefix' ∪ {j}
17         end
18     end
19     if Suffixes'_Prefix' ≠ ∅ then
20         Answer = Answer ∪ {⟨Prefix', IA_Prefix', Suffixes'_Prefix'⟩}
21     end
22 end
23 return Answer
```

En el algoritmo 5.2 se muestra el pseudocódigo de la función *ECGenAndCountDin*. En las líneas de la 3 a la 5 del pseudocódigo de la función *ECGenAndCountDin* se inicializa cada nueva clase de equivalencia. Posteriormente en las líneas 6 a la 14 se verifica cada conjunto de ítems de la clase de equivalencia de entrada EC_{k-1}, en la línea 7 se calcula el soporte de forma similar a como se calcula en el algoritmo *CA*, en las líneas de la 8 a la 10 se actualizan los FI y en la línea 13 se verifica el soporte global de los nuevos conjuntos candidatos. La función *Rerun* obtiene los arreglos de enteros asociados a cada uno de los ítems que forman el conjunto candidato a verificar, y calcula su soporte realizando operaciones *AND*. Si los conjuntos de ítems verificados son frecuentes se adicionan a la clase de equivalencia actual (línea 14). Si la clase de equivalencia actual, al terminar de construirse, tiene al menos un elemento, entonces se adiciona al conjunto de clases de equivalencia devuelto por la función (líneas 15 y 16).

Para la eliminación de transacciones, el algoritmo recibe como entrada una lista con los identificadores de las transacciones que se desea eliminar y el pseudocódigo de la función

ECGenAndCountDin sólo cambia en la línea 9. El caso de una modificación se puede reducir a una eliminación seguida de una adición.

5.2. Experimentación y resultados

Debido a que no fue posible conseguir otros algoritmos que calculen todos los FI en conjuntos de datos dinámicos y además, a que implementar los algoritmos propuestos por otros autores puede ser muy complicado y no hay garantía de que sean todo lo eficientes que dicen los artículos, se realizaron dos experimentos que dan una idea de cómo escala el algoritmo al adicionar y eliminar conjuntos de transacciones. Los resultados se compararon con algoritmos que calculan todos los FI en conjuntos de datos estáticos.

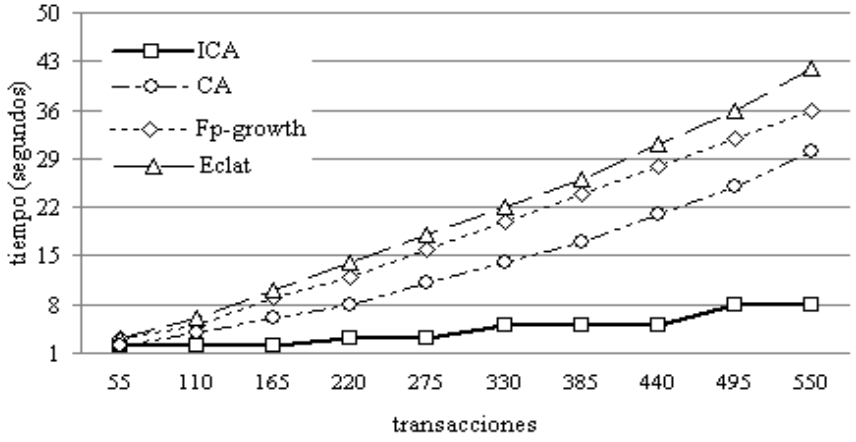

Fig. 19. Escalamiento al adicionar transacciones con soporte 0,02 (*El País*)

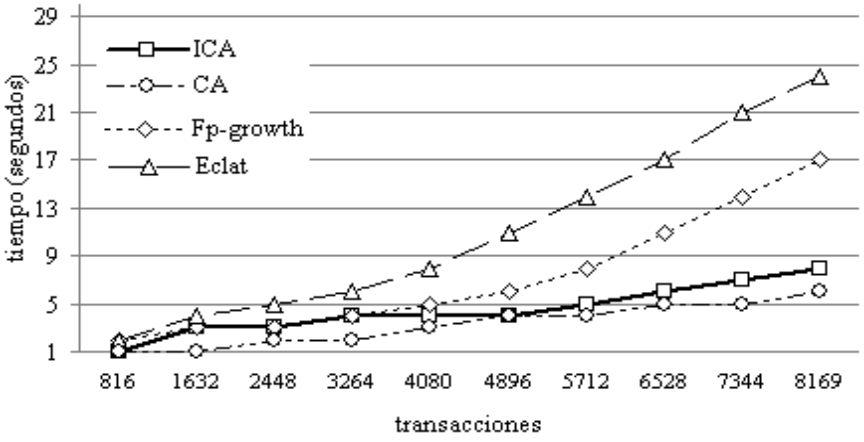

Fig. 20. Escalamiento al adicionar transacciones con soporte 0,02 (*TDT*)

Para experimentar se utilizaron 4 de los conjuntos de datos descritos en la sección anterior, éstos son *El País*, *TDT*, *Kosarak* y *Webdocs*. Cada uno de los conjuntos de datos se dividió aleatoriamente en 10 conjuntos de igual tamaño, la comparativa se realizó entre los algoritmos *Eclat*, FP-growth, *CA* e *ICA*.

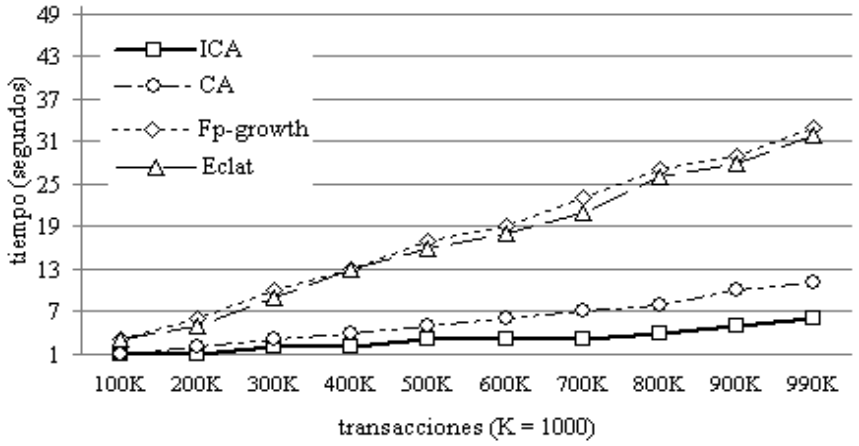

Fig. 21. Escalamiento al adicionar transacciones con soporte 0,002 (*Kosarak*).

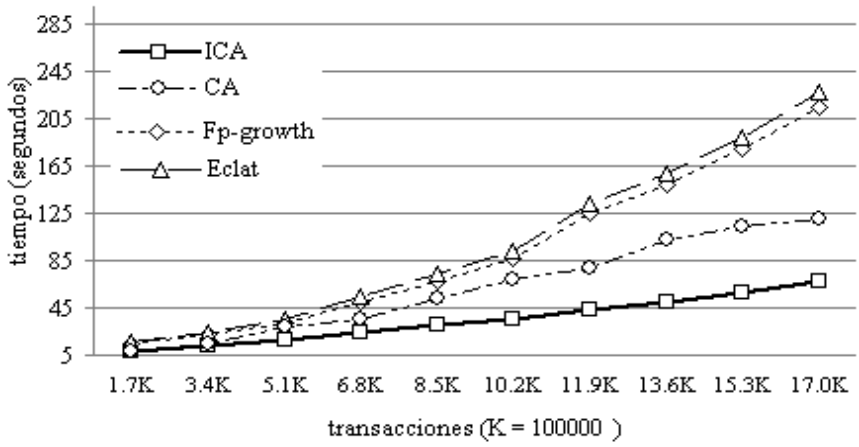

Fig. 22. Escalamiento al adicionar transacciones con soporte 0,13 (*Webdocs*).

En el primer experimento se ejecutaron los 3 algoritmos para datos estáticos sobre la acumulación de los 10 conjuntos de datos, de forma similar al último experimento de la sección 4. En el caso del algoritmo ICA, se fue adicionando cada uno de los 10 conjuntos de datos; de esta forma los primeros algoritmos siempre procesan el conjunto de datos desde el principio mientras *ICA* reutiliza los FI calculados hasta el momento para calcular los nuevos FI. Debido a que la división de los conjuntos de datos se realizó de forma aleatoria, el experimento se ejecutó 10 veces y se tomaron los promedios de los tiempos de ejecución.

Como se puede observar en las figuras de la 19 a la 22, el algoritmo *ICA* escala mejor que el resto de los algoritmos, sólo con el conjunto de datos *TDT* (Figura 20) el algoritmo *CA* tiene un mejor desempeño. Esto se debe a que la cantidad de conjuntos candidatos depende de la distribución que tengan los ítems de mayor soporte en el conjunto de transacciones que se adicione. Tanto los algoritmos *CA* como *ICA* tienen las mismas características, al adicionarse un conjunto de transacciones con una gran densidad de ítems con soportes

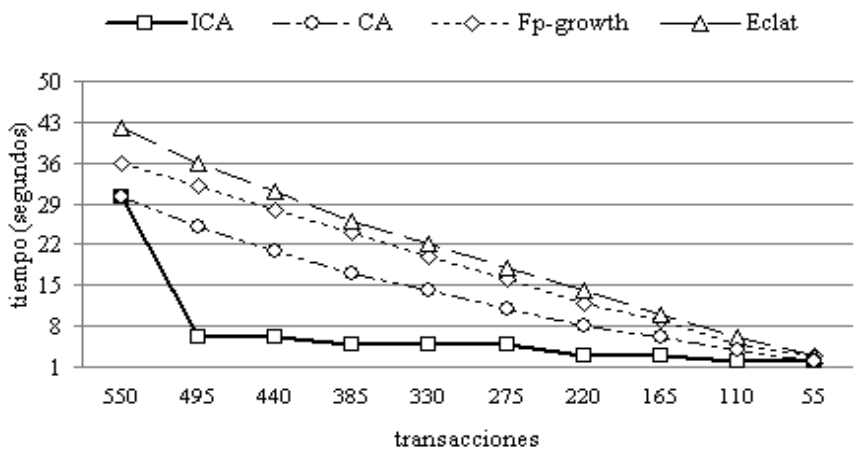

Fig. 23. Escalamiento al eliminar transacciones con soporte 0,02 (*El País*)

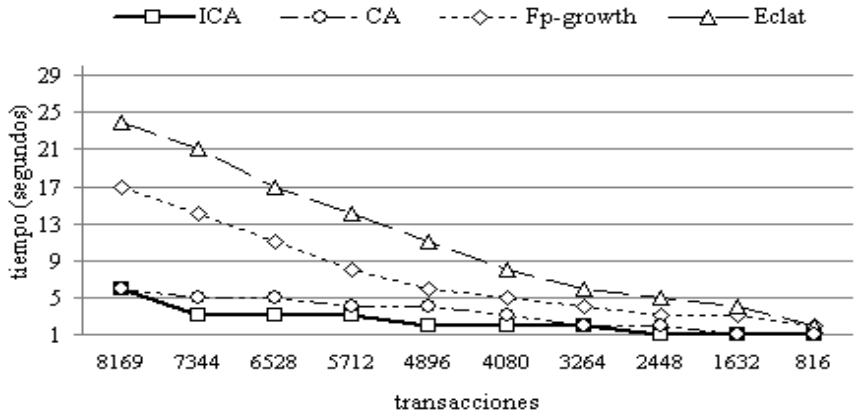

Fig. 24. Escalamiento al eliminar transacciones con soporte 0,02 (*TDT*)

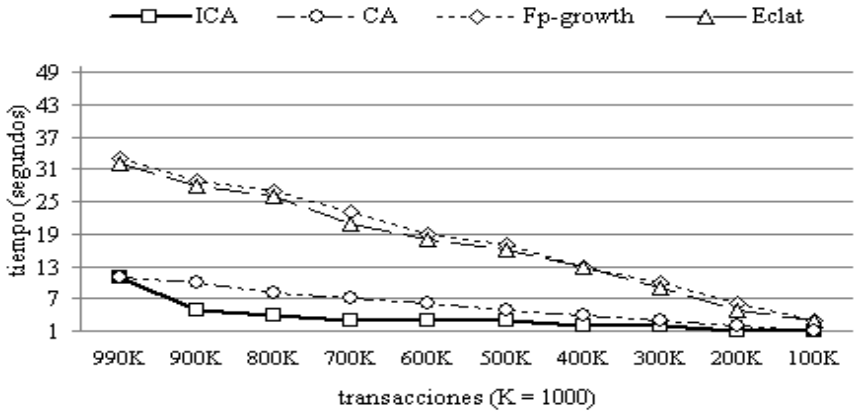

Fig. 25. Escalamiento al eliminar transacciones con soporte 0,002 (*Kosarak*)

altos, el algoritmo *ICA* necesita actualizar muchos candidatos siendo menos eficiente que el algoritmo *CA*. Por tanto, en el caso de adicionarse conjuntos de transacciones con muchos ítems de soporte global alto, es más conveniente utilizar el algoritmo *CA*.

El segundo experimento se realizó en sentido inverso, se comenzó con el conjunto de datos completo y se fueron eliminando los mismos conjuntos de transacciones utilizados en el primer experimento. En este experimento el algoritmo ICA tuvo los mejores resultados en todos los conjuntos de datos, al comenzar con todo el conjunto de datos e ir eliminando transacciones se da menos la situación de que aparezcan nuevos FI. Los resultados del segundo experimento se pueden observar en las figuras de la 23 a la 26.

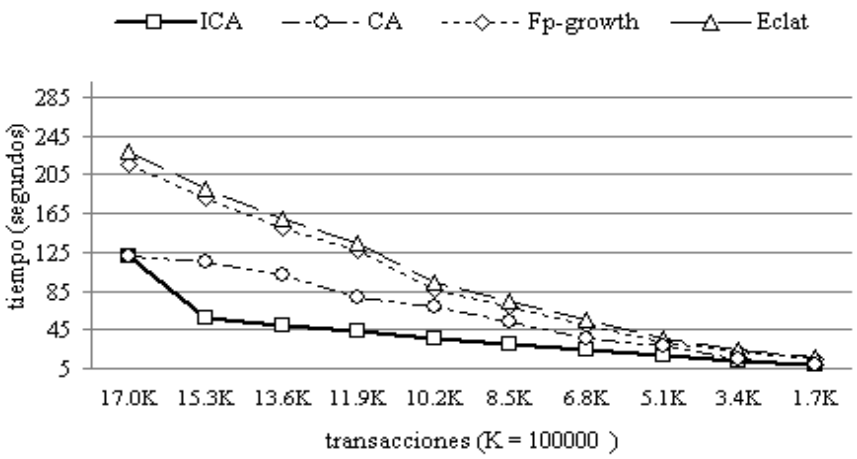

Fig. 26. Escalamiento al eliminar transacciones con soporte 0,13 (*Webdocs*)

El algoritmo *ICA* no supone que todo el conjunto de datos cabe en memoria, dificultad presente en los algoritmos anteriores. Los experimentos realizados muestran que, después de una modificación en el conjunto de datos, es más eficiente reutilizar los FI previamente calculados para calcular los nuevos FI que procesar todo el conjunto de datos desde el principio.

6. Conclusiones y trabajo futuro

En el presente reporte se presentaron dos algoritmos para calcular todos los FI, uno para conjuntos de datos estáticos (*CA*) y otro para conjuntos de datos dinámicos (*ICA*).

El algoritmo *CA*, desarrollado para conjuntos de datos estáticos, se comparó con los mejores algoritmos reportados en la literatura, los códigos fuente de estos algoritmos fueron facilitados por los autores y los experimentos se realizaron bajo las mismas condiciones de *hardware* y *software*. El algoritmo *CA* obtuvo mejores resultados en la mayoría de los conjuntos de datos, sólo fue superado por *PatriciaMine* en dos conjuntos de datos con los valores más pequeños del soporte. Además se evaluó el escalamiento del algoritmo obteniéndose los mejores resultados en todos los casos.

En el caso del algoritmo *ICA*, desarrollado para conjuntos de datos dinámicos, no se pudieron obtener otros algoritmos para comparar. Para evaluar el desempeño de este algoritmo se dividió cada conjunto de datos, aleatoriamente, en 10 partes y se graficó su escalabilidad después de adicionar y eliminar cada parte. Con este experimento se mostró que el algoritmo *ICA* es más conveniente para procesar conjuntos de datos dinámicos que los mejores algoritmos reportados para conjuntos de datos estáticos; esto se debe a que reutiliza los FI calculados previamente para calcular los nuevos FI. Además, *ICA* no presupone que el conjunto de datos cabe en memoria, como lo hacen los demás algoritmos para datos dinámicos reportados en la literatura.

Los resultados de este trabajo se reportan en [25].

Con base en los resultados obtenidos en este reporte, existen algunas variantes que se pueden seguir como posibles trabajos futuros.

Primeramente, cuando los algoritmos para el cálculo de los FI se aplican con umbrales de soporte muy bajos o cuando los conjuntos de datos tienen miles de ítems diferentes, la cantidad de FI calculados está en el orden de los millones. Una gran cantidad de FI puede afectar el desempeño de otras aplicaciones que necesiten de éstos. Una posible línea a seguir es adaptar los algoritmos presentados para calcular sólo los FI maximales.

Otra posible línea es tratar el problema de la minería de flujos, en la cual constantemente está llegando información y no se puede recorrer dos veces el conjunto de datos para calcular los FI.

Como tarea inmediata, se debe realizar una comparación más precisa del algoritmo *ICA*, para ello se necesita conseguir el código fuente de otros algoritmos para datos dinámicos.

Referencias bibliográficas

1. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *In Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, pages 207–216, 1993.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *In Proceedings of the 20*th *International Conference on Very Large Data Bases, VLDB'94, Santiago de Chile, Chile*, pages 487–499, 1994.
3. S. Ahmed, F. Coenen, and P. Leng. Tree-based partitioning of data for association rule mining. *Knowledge and Information Systems Journal*, 10(3):315–331, 2006.
4. C. Borgelt. Efficient implementations of apriori and eclat. *In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementation (FIMI), Melbourne, Florida, USA*, 2003.
5. D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. *In Proceedings of the International Conference on Data Engineering (ICDE), Heidelberg, Germany*, 2001.
6. T. Calders, N. Dexters, and B. Goethals. Mining frequent itemsets in a stream. *In Proceedings of the IEEE International Conference on Data Mining, Omaha, Nebraska, USA*, pages 83–92, 2007.
7. T. Calders, N. Dexters, and B. Goethals. Mining frequent items in a stream using flexible windows. *Intelligent Data Analysis*, 12(3), 2008.
8. M. S. Chen, J. Han, and P. S. Yu. Data mining: An overview from a database perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, 1996.
9. D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. *In Proceedings of the International Conference on Data Engineering (ICDE), New Orleans, LA*, pages 106–114, 1996.
10. D. W. Cheung, S. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. *In Proceedings of the International Conference on Data Engineering (ICDE), Birmingham U.K.*, 1997.

11. D. W. Cheung and O. R. ZaIane. Incremental mining of frequent patterns without candidate generation or support constraint. *In Proceedings of the 7th International Database Engineering and Applications Symposium (IDEAS 2003)*, pages 111–116, 2003.
12. C. K. Chui and B. Kao. A decremental approach for mining frequent itemsets from uncertain data. *In Proceedings of the 12th Pacific-Asia Conference (PAKDD 2008)*, pages 64–75, 2008.
13. U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. *Advances in Knowledge Discovery and Data Mining, AAAI Press*, pages 1–34, 1996.
14. R. P. Gopalan and Y. G. Sucahyo. High performance frequent patterns extraction using compressed fp-tree. *In Proceedings of the SIAM International Workshop on High Performance and Distributed Mining, Orlando, USA*, 2004.
15. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *In Proceedings ACM-SIGMOD International Conference on Management of Data, New York, NY, USA*, 2000.
16. R. A. García Hernández, J. Fco. Martínez Trinidad, and J. A. Carrasco Ochoa. A fast algorithm to find all the maximal frequent sequences in a text. *9th Iberoamerican Congress on Pattern Recognition, Lecture Notes in Computer Science*, 3287:478–486, 2004.
17. R. A. García Hernández, J. Fco. Martínez Trinidad, and J. A. Carrasco Ochoa. A new algorithm for fast discovery of maximal sequential patterns in a document collection. *7th Intelligent Text Processing and Computational Linguistics, Lecture Notes in Computer Science*, 3878:514–523, 2006.
18. J. Hipp, U. Güntzer, and G.Ñakhaeizadeh. Algorithms for association rule mining – a general survey and comparison. *ACM SIGKDD, In Proceedings of the International Conference on Knowledge Discovery and Data Mining, Boston, MA, USA*, 2000.
19. J. D. Holt, M. S. Chen, and P. S. Yu. Mining association rules in text databases using multipass with inverted hashing and pruning. *In Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, Washington, DC, USA*, pages 49–56, 2002.
20. J. D. Holt and S. M. Chung. Efficient mining of association rules in text databases. *In Proceedings of the Eigth International Conference on Information and Knowledge Management, ACM, Kansas City, Missouri*, 1999.
21. J. D. Holt and S. M. Chung. Multipass algorithms for mining association rules in text databases. *Knowledge and Information Systems, Springer-Verlag*, 3(2):168–183, 2001.
22. J. D. Holt and S. M. Chung. Mining association rules using inverted hashing and pruning. *Information Processing Letters*, (4):211–220, 2002.
23. B. Kalpana and R.Ñadarajan. Incorporating heuristics for efficient search space pruning in frequent itemset mining strategies. *Current Science 94*, pages 97–101, 2008.
24. J. L. Koh and S. F. Shieh. An efficient approach for maintaining association rules based on adjusting fp-tree structures. *In Proceedings of the 9th International Conference on Database Systems for Advanced Applications (DASFAA 2004)*, 2973:417–424, 2004.
25. R. Hernández León, J. Hernández Palancar, J. A. Carrasco Ochoa, and J. F. Martinez Trinidad. A novel incremental algorithm for frequent itemsets mining in dynamic datasets. *In Proceedings of the 13th Iberoamerican Congress on Pattern Recognition, La Habana, Cuba*, 2008.
26. C. K-S. Leung, M. Anthony, F. Mateo, and D. A. Brajczuk. A tree-based approach for frequent pattern mining from uncertain data. *In Proceedings of the 12th Pacific-Asia Conference (PAKDD 2008)*, pages 653–661, 2008.
27. C. K-S. Leung, Q. I. Khan, and T. Hoque. Cantree: A tree structure for efficient incremental mining of frequent patterns. *In Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005)*, pages 274–281, 2005.
28. J. Hernández Palancar, R. Hernández León, J. E. Medina Pagola, and A. Hechavarría Díaz. Mining frequent patterns using compressed vertical binary representations. *In Proceedings of the Workshop Foundation of Semantic Oriented Data and Web Mining, IEEE International Conference on Data Mining, Houston, Texas, USA*, 2005.
29. J. S. Park, M. S. Chen, and P. S. Yu. Using a hash-based method with transaction trimming and database scan reduction for mining association rules. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):813–825, 1997.
30. A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. *In Proceedings of the Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA*, 2003.
31. A. Savasere, E. Omiecinski, and S.Navathe. An efficient algorithm for mining association rules in large databases. *Technical Report GIT-CC-95-04, Institute of Technology, Atlanta, USA*, 1995.

32. H. Schmid. Probabilistic part-of-speech tagging using decision trees. In *International Conference on New Methods in Language Processing, (Software in: www.ims.uni-stuttgart.de/ftp/pub/corpora/tree-tagger1.ps.gz)*, Manchester, UK, 1994.
33. P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. *In Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, USA*, 2000.
34. Y. G. Sucahyo and R. P. Gopalan. Ct-itl: Efficient frequent item set mining using a compressed prefix tree with pattern growth. *In Proceedings of 14th Australasian Database conference, Adelaide, Australia*, 2003.
35. Y. G. Sucahyo and R. P. Gopalan. Ct-pro: A bottom-up non recursive frequent itemset mining algorithm using compressed fp-tree data structure. *In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementation (FIMI). Brighton UK*, 2004.
36. S. K. Tanbeer, C. F. Ahmed, B. S. Jeong, and Y. K. Lee. Cp-tree: A tree structure for single-pass frequent pattern mining. *In Proceedings of the 12th Pacific-Asia Conference (PAKDD 2008)*, pages 1022–1027, 2008.
37. T. Uno and H. Arimura. Ambiguous frequent itemset mining and polynomial delay enumeration. *In Proceedings of the 12th Pacific-Asia Conference (PAKDD 2008)*, pages 357–368, 2008.
38. M. J. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. *In Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures, Newport, RI, USA*, 1997.
39. M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. *In Proceedings of the 3rd International Conference on KDD and Data Mining, EU*, 1997.

RT_006, agosto 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