RNPS No. 2143
ISSN 2072-6260
Versión Digital

SERIE GRIS

REPORTE TÉCNICO
Minería de Datos

Obtención de conjuntos frecuentes usando cómputo reconfigurable

Ing. Alejandro Mesa Rodríguez, Dr. C. José Hernández Palancar, Dr. C. Claudia Feregrino Uribe, Dr. C. René Cumplido Parra

RT_008 **Diciembre 2009**

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

Obtención de conjuntos frecuentes usando cómputo reconfigurable

Ing. Alejandro Mesa Rodríguez, Dr. C. José Hernández Palancar, Dr. C. Claudia Feregrino Uribe, Dr. C. René Cumplido Parra

RT_008 **Diciembre 2009**

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

1. Introducción 2
2. Aspectos generales de conjuntos frecuentes 4
   2.1 Características de los métodos de generación de itemsets 5
   2.2 Representación de los itemsets 6
3. Métodos de obtención de conjuntos frecuentes 7
   3.1 Métodos tipo Apriori 7
   3.2 Hashing 8
   3.3 Particionamiento 11
   3.4 Estructuras Arbóreas 12
4. Computo reconfigurable 14
   4.1 Conceptos Básicos 14
   4.2 FPGAs 15
   4.3 Flujo de diseño 16
5. Implementación de métodos de obtención de conjuntos frecuentes sobre FPGAs 17
   5.1 Apriori 18
   5.2 DHP 21
   5.3 FP-Growth 22
6. Conclusiones 24
Referencias bibliográficas 25

Obtención de conjuntos frecuentes usando cómputo reconfigurable

Ing. Alejandro Mesa Rodríguez, Dr. C. José Hernández Palancar, Dr. C. Claudia Feregrino Uribe, Dr. C. René Cumplido Parra

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

RT_008 CENATAV
Fecha del camera ready: 31 de julio de 2009

Resumen. El descubrimiento de Conjuntos Frecuentes (FI por sus siglas en inglés) en bases de datos es una de las técnicas de más reciente desarrollo dentro de la Minería de Datos. En esta área se han desarrollado muchos algoritmos. Estos, al enfrentarse a bases de datos muy extensas no presentan un buen rendimiento o son incapaces de procesarlas. Por esto surge la necesidad de buscar alternativas que mejoren el rendimiento del descubrimiento de FI en Bases de Datos extensas. Una posible línea a seguir para enfrentar estos problemas es el empleo de arquitecturas hardware para la aceleración de la ejecución de algoritmos de este tipo. En este trabajo se realiza un estudio del estado del arte de las arquitecturas hardware propuestas para enfrentar este tipo de problemas.

Palabras Claves: minería de datos, conjuntos de ítems frecuentes, aceleración por hardware, cómputo reconfigurable

Abstract. Mining frequent itemsets (FI) in large databases is a wide used technique in Data Mining. Several algorithms were developed for this, although in many environments, the execution of those algorithms, takes more time and resources than they are expected to or simply can't achieve the task and consequently they fail. This is why finding more efficient alternatives for those algorithms is an active topic. One used approach to address these problems is the use of specific hardware architectures for accelerating the task. This technical report presents the state of the art of specialized hardware architectures to address such problems.

Keywords: Data Mining, Frequent Itemsets, Hardware Acceleration, Reconfigurable Computing

1. Introducción

Hoy en día los sistemas informáticos están involucrados en múltiples áreas de la vida. Es un reto manejar toda la información que es emitida y almacenada por la creciente cantidad de fuentes de datos que estos generan. Información sobre transacciones de ventas, trazas de sistemas y sitios webs, emisiones de noticias, publicaciones seriadas, transacciones de tarjetas de crédito y mediciones de parámetros en las industrias entre muchos otros, generan cientos de terabytes de datos en el transcurso de un día. Toda esta información sobrepasa la capacidad de asimilación de las personas, surgiendo por este motivo muchas técnicas de minería de datos para tratar este problema. Estas técnicas incluyen desde la extracción de información útil no trivial, como son el minado de patrones frecuentes y la extracción de reglas de asociación, hasta la organización de los datos para su posterior uso (ej. clasificación, agrupamiento, etc.). Todas estas técnicas están destinadas a extraer conocimiento potencialmente útil y mostrarlo de manera comprensible para el ser humano.

La obtención de conjuntos frecuentes para el descubrimiento de vínculos o asociaciones en estos datos es una técnica de reciente desarrollo dentro de la minería de datos. Esta se evidencia con fuerza con la aparición del algoritmo Apriori [1] introducido por Rakesh Agrawal y Ramakrishnan Srikant, y aunque ya existían esfuerzos en el tema, este revolucionó el área del descubrimiento de asociaciones y sirvió de base para infinidad de algoritmos que surgieron posteriormente. El descubrimiento de conjuntos frecuentes es una técnica que es computacionalmente compleja por el cúmulo de operaciones simples que tiene que repetir sobre el conjunto de datos, y requiere de grandes recursos computacionales y de mucho tiempo para resolver la explosión combinatoria que se genera.

Muchos de estos algoritmos, al tratar con conjuntos de datos muy grandes y dispersos (como lo son las colecciones de documentos) no dan una respuesta en un tiempo aceptable o simplemente no son capaces de manejarlos. Esto ocurre fundamentalmente por la presencia de miles de elementos diferentes o por el uso de un umbral de soporte (minsup[1]) muy bajo. Cuando esta cota de soporte mínima es cercana a cero, dado una n cantidad de elementos diferentes, la cantidad de conjuntos frecuentes que tiene que procesar el algoritmo tiende a 2^n. Esto provoca que la complejidad de este tipo de algoritmos sea exponencial.

Por estos motivos, en la actualidad, los esfuerzos en esta área van dirigidos a la mejora de la eficiencia para reducir los tiempos de respuesta y poder satisfacer la demanda existente de velocidad de procesamiento. El uso de dispositivos hardware para la disminución del tiempo de ejecución de esta tarea, es un tema que se está explorando en los últimos años, siendo en la actualidad un área activa de investigación. Además, cada vez aparecen más trabajos relacionados con la aceleración de la ejecución de algoritmos utilizando arquitecturas hardware, evidenciando la tendencia a usar este tipo de estrategias.

Los sistemas basados en dispositivos hardware proporcionan un menor tiempo de procesamiento que sus correspondientes implementaciones en software. Estos explotan el hecho de que la mayor parte del tiempo de procesamiento para tareas de computación intensivas es gastado en una porción relativamente pequeña del código. Un incremento en la velocidad de esta porción mediante la utilización de hardware, puede mejorar su rendimiento por encima del que presenta al ejecutarse en un microprocesador de propósito general.

Un FPGA es un dispositivo programable de propósito general que integra una gran cantidad de dispositivos lógicos programables en un chip. Su tamaño y velocidad es equiparable a los ASICs[2], pero son más flexibles y su ciclo de diseño es más corto. Los sistemas basados en FPGAs proporcionan un mejor desempeño que sus correspondientes implementaciones en software, comportándose, en la mayoría de los casos, desde 10 y hasta 1000 veces más rápido, dependiendo generalmente de la cantidad de operaciones simples y de la dependencia de datos que exista entre las operaciones. Los de mayor número de operaciones simples son los que presentan mayor aceleración al implementarlos sobre estos dispositivos.

A pesar del reto que representa diseñar arquitecturas eficientes que manejen la complejidad que aportan características presentes en este tipo de algoritmos como el control intensivo sobre los datos

---

1 Es la mínima cantidad de apariciones en la base de datos que debe tener un conjunto de elementos para que se considere como frecuente.

2 Circuitos Integrados para Aplicaciones Específicas.

y el manejo de grandes volúmenes de estos, el potencial de mejora en velocidad de procesamiento justifica el esfuerzo de diseño de estas [2-6].

Cada día va en aumento la cantidad de información digital que se genera y la cantidad de fuentes que la emiten y aunque la capacidad de cómputo de las computadoras de propósito general también ha tenido notables avances, no es comparable con la demanda de datos a procesar y tiempos de respuestas que se necesitan para determinadas aplicaciones.

Este trabajo se enfoca en el minado de patrones frecuentes, en particular en la aceleración del minado de conjuntos frecuentes de ítems utilizando hardware reconfigurable. En general existen varias implementaciones de algoritmos eficientes para determinar conjuntos frecuentes que se basan fundamentalmente en el uso de operaciones lógicas como ANDs, XORs y conteo de unos sobre un arreglo de bits. Este tipo de operaciones son muy convenientes de implementar sobre FPGA, de ahí la factibilidad que ofrece el desarrollo de este tema.

Trabajos previos se basaron en arquitecturas de arreglos sistólicos o pipelines, representando los conjuntos de ítems como listas de ítems, no haciendo explicita la ausencia del resto de los ítems que conforman la base de datos, pero utilizando mayor cantidad de bits para cada representación (por lo general 32 bits por ítem). También se encontró la tendencia a reducir la cantidad de veces que se recorría la base de datos, eliminando de ella elementos, que según alguna heurística, no le aportaba información para obtener conjuntos frecuentes, ya sea elementos de una transacción, reduciendo su tamaño o la transacción completa. En estos diseños corresponde la mayor aceleración a la menor cantidad de datos que procesen.

2. Aspectos generales de conjuntos frecuentes

Los algoritmos de obtención de conjuntos frecuentes están orientados al descubrimiento de dependencias o vínculos entre conjuntos de atributos, variables o (como suele denominarse en este campo) ítems. Los primeros tipos de aplicaciones en el que se comprobó la utilidad de este tipo de técnicas fueron los sistemas que procesan los datos de ventas obtenidos mediante lectores de código de barra, conocidos como Datos de la Canasta [7]. Un registro o tupla en estos datos está compuesto por la fecha de la transacción y los ítems comprados en ella.

Estas técnicas pueden ser aplicadas a cualquier tipo de dato, existiendo diversos trabajos orientados a los diferentes tipos, e incluso a la mezcla de ellos. En el caso de aplicarse a noticias, o datos textuales en general, un registro o tupla sería cada documento a procesar, mientras que los ítems serían los términos significativos presentes en los documentos. En este caso, la colección de datos puede considerarse con similar formato a los de la forma clásica; o sea, como un conjunto de transacciones compuestas de ítems.

Un conjunto de ítems, o itemset, es aquel cuyos elementos son atributos, variables o campos de cualquier base de datos. Aunque no es un requisito esencial, los ítems que se considerarán en lo sucesivo son aquellos que toman valores bivalentes. El tamaño de un itemset, o conjunto de ítems, está dado por la cantidad de ítems contenidos en el itemset, lo que se define como su cardinalidad, definiéndose un k-itemset como un itemset de cardinalidad k.

El cubrimiento (transaccional) de un itemset X en una base de datos D está formado por el conjunto $D(X) = \{T \in D | X \subseteq T\}$.

El soporte (Support) de un itemset o conjunto de ítems X en una base de datos D está dado por la cantidad de transacciones en que aparece el elemento (|D(X)|) entre la cantidad de transacciones de la BD.

Un conjunto de ítems (itemset) frecuente o extenso es aquel que ocurre con un soporte no menor que un mínimo dado (minsup).

2.1 Características de los métodos de generación de itemsets

Generar todos los itemsets frecuentes es una tarea ardua y de alto costo. En general, la evaluación de los subconjuntos de un conjunto de ítems que caracterizan a una base de datos se corresponde con la búsqueda en el espacio de estado asociado con el retículo que estos forman. [8].

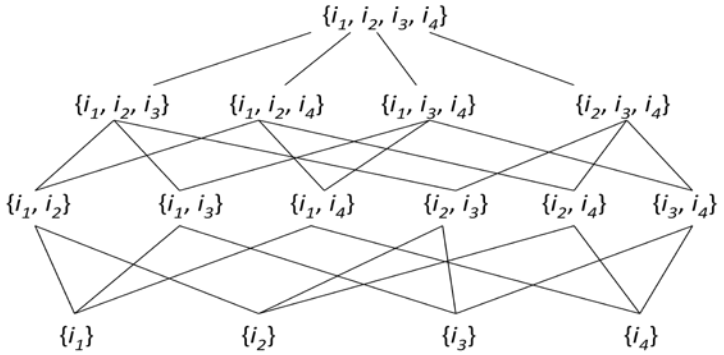

Fig. 1. Espacio de búsqueda de conjuntos frecuentes para 4 ítems.

Los soportes de los itemsets de un retículo disminuyen al recorrerlos desde los 1-itemsets hasta el itemset universo, verificándose la clausura descendente de sus soportes. Esta propiedad plantea que el soporte de cualquier subconjunto de un itemset es mayor o igual que el soporte de ese itemset. Además, todo subconjunto de un itemset frecuente es frecuente, mientras que cualquier supraconjunto de un itemset no frecuente tampoco es frecuente.

Considerando la propiedad de la clausura descendente de los soportes, cualquiera sea el minsup, todo retículo posee un borde o frontera que separa los itemsets frecuentes de los no frecuentes. Este borde se asocia con los denominados itemsets maximales. Un conjunto de ítems (itemset) es maximal si no es un subconjunto propio de ningún otro itemset frecuente.

Otro tipo de itemset utilizado por algunos algoritmos es el conocido como Conjunto de Ítems Cerrado, el cual depende fuertemente de las características de las transacciones de la base de datos. Estos son todos los itemsets X de una BD tal que no existe otro conjunto diferente Y tal que $X \subset Y$, siendo $D(X) = D(Y)$, verificándose además la expresión: $X = \cap D(X)$.

En general, un itemset cerrado es el de mayor cardinalidad en cualquier secuencia de igual soporte en el retículo. El conjunto de los itemsets maximales es un subconjunto de los itemsets cerrados, cualquiera sea la base de datos.

Los algoritmos de generación de itemsets establecen un orden de recorrido del espacio de itemsets. Si se considera un orden lexicográfico entre los ítems y estos se ordenen según este orden entonces se pueden agrupar por clases de equivalencia. La clase de equivalencia de un itemset X, indicado por E(X), está dado por el siguiente conjunto de itemsets:

$$E(X) = \{Y | Prefix_{k-1}(Y) = X \wedge |Y| = K\}$$

siendo $Prefix_k$ el prefijo de tamaño k de c, formado por sus k primeros ítems, según orden lexicográfico.

Considerando lo anterior, los métodos de generación de los itemsets frecuentes definen diferentes recorridos. Estos métodos pueden clasificarse atendiendo a la dirección y amplitud de los recorridos. Si se considera la dirección del recorrido, estos pueden clasificarse como:

- Métodos descendentes: Si el recorrido se realiza desde las raíces (1-itemsets) hasta los itemsets fronteras o maximales.
- Métodos ascendentes: Si el recorrido se realiza en el sentido opuesto, desde supraconjuntos de los itemsets fronteras (o maximales) hasta los itemsets fronteras (o maximales).
- Si se considera la amplitud de los recorridos, estos pueden clasificarse como:
- Métodos a lo ancho: Si los itemsets de igual tamaño se evalúan antes de evaluar otro de diferente tamaño.
- Métodos en profundidad: Si se evalúan los itemsets por cada rama del árbol. Representación de los itemsets.

2.2 Representación de los itemsets

La forma de representar los itemsets es muy importante en el cálculo de los conjuntos frecuentes. Conceptualmente, las bases de datos son matrices bidimensionales, donde las filas representan las transacciones y las columnas los ítems existentes. Estas matrices pueden implementarse de cuatro formas diferentes[9]:

- Vector Horizontal de Ítems (HIV): Las transacciones se organizan como un conjunto de filas, donde cada fila almacena el identificador de la transacción (TID) y un vector de bits, para representar por cada ítem su presencia (con uno) o ausencia (con cero) en la transacción.
- Lista Horizontal de Ítems (HIL): Esta representación es similar a HIV exceptuando que cada fila almacena una lista ordenada de identificadores de ítems (IID), representando sólo los ítems que están presentes en la transacción.
- Vector Vertical de TID (VTV): Las transacciones se representan como un conjunto de columnas, asociadas con los ítems, donde por cada columna se almacena su IID y un vector de bits para representar la presencia ó ausencia del ítem en cada transacción. Se puede notar que VTV ocupa exactamente la misma cantidad de memoria que la representación HIV respecto a los vectores de bits, aunque por lo general se precisa menos memoria para almacenar los identificadores asociados.
- Lista Vertical de TID (VTL): Esta representación es similar a VTV exceptuando que cada columna almacena una lista ordenada de TID (tidList), representando sólo las transacciones en las cuáles el ítem está presente.

3. Métodos de obtención de conjuntos frecuentes

3.1 Métodos tipo Apriori

Entre los métodos descendentes con recorrido a lo ancho se encuentra el método paradigmático de esta clasificación es el algoritmo Apriori [1]. La estrategia que este algoritmo fue la base para muchos de los algoritmos desarrollados posteriormente.

La estrategia que este algoritmo sigue es del tipo descendente a lo ancho, utilizándose como base de muchos otros algoritmos desarrollados hasta la fecha. Este algoritmo está compuesto por 3 secciones: generación de candidatos, poda y conteo de soporte. Estas secciones se repiten hasta que ya no haya más conjuntos frecuentes. Los 1-itemsets frecuentes son obtenidos y servidos al sistema iniciando el proceso.

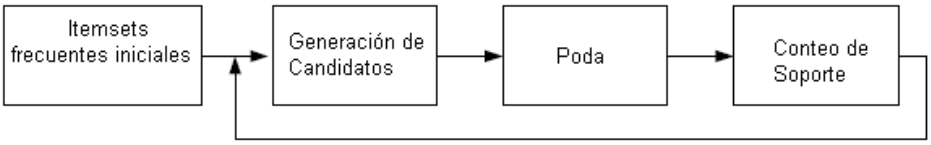

Fig. 2. Estructura funcional del algoritmo Apriori.

Para la generación de candidatos de nivel k (k-itemset), se hace uso de la información referida a los conjuntos frecuentes de la iteración anterior (k-1 itemset frecuentes) generándose según el siguiente procedimiento:

$\forall\, c_1, c_2 \in C_{k-1}\ do$

$with\ c_1 = (i_1, \cdots, i_{k-2}, i_{k-1})$

$and\ c_1 = (i_1, \cdots, i_{k-2}, i^*_{k-1})$

$and\ i_{k-1} < i^*_{k-1}$

$c \coloneqq\ c_1 \cup c_2 = (i_1, \cdots, 2, i_{k-1}, i^*_{k-1})$

La próxima etapa del algoritmo es la poda, donde se garantiza que no pase al conteo de soporte itemsets que se conocen a priori que no van a ser frecuentes, eliminando a todos aquellos candidatos que no cumplan con que todos sus subconjuntos sean también frecuentes. Como este k-itemset se generó a partir de k-1 itemsets que eran frecuentes, solo hay que verificar los subconjuntos de tamaño k-1 según el siguiente procedimiento:

$\forall\, c \in C_k\ do$

$\forall\, i \in c :\ c - \{i\} \in C_{k-1}$

La tercera etapa del algoritmo es el conteo del soporte. En esta se cuenta la cantidad de apariciones de los k-itemsets candidatos en la base de datos y se determina, según el soporte mínimo establecido, el conjunto de los k-itemsets frecuentes. Esto se realiza según el siguiente procedimiento:

$\forall\, t \in T\ do$

$\forall\, c \in C\ do$

$if\ c\ \subseteq t$

$suport(c) + +$

Estas 3 etapas se repiten hasta que a la salida de la 3ra no haya ningún itemset frecuente. La salida del algoritmo es el conjunto de todos los itemsets frecuentes de todos los niveles.

De forma general, la mayoría de los algoritmos tipo Apriori primero construyen un conjunto de itemsets candidatos basados en alguna heurística y, posteriormente, determinan el subconjunto que realmente contiene los itemsets frecuentes. Este proceso puede realizarse de forma repetitiva conociendo que los itemsets frecuentes obtenidos en una iteración servirán de base para la generación del conjunto candidato en la siguiente iteración.

Las heurísticas que siguen estos algoritmos están basadas fundamentalmente en la propiedad de clausura descendente y la usan tanto para generar los itemsets candidatos como para podar el espacio de búsqueda de los itemsets frecuentes. Una vez obtenidos los candidatos de un nivel k, se procede a contar el soporte de estos itemsets para obtener los frecuentes. Con los frecuentes de nivel k se pasa a calcular los frecuentes de nivel k+1.

3.2 Hashing

En general los algoritmos descendentes con recorrido a lo ancho generan un conjunto de itemsets candidatos, basados en alguna heurística y a partir de éste determinan el subconjunto que contiene a los itemsets frecuentes contando en la colección de datos la cantidad de veces que aparecen. En estos casos la heurística para decidir si un itemset es candidato es crucial para el funcionamiento de estos algoritmos. De esta forma, para lograr una mayor eficiencia se debería contar con heurísticas que generen solamente candidatos con una alta probabilidad de ser frecuentes. Debido a esto, han surgido algoritmos con heurísticas que mejoran el cálculo de los conjuntos de itemset candidatos utilizando tablas de hash. Estos algoritmos utilizan tablas de hash para obtener información sobre las transacciones y determinar si esta o sus ítems son relevantes para próximas iteraciones. Con esta información estos algoritmos logran la eliminación tanto de aquellas transacciones que no serán capaces de contener a itemsets mayores, así como de los ítems que contienen las transacciones y que no formarán parte de ningún itemset frecuente de tamaño superior. Esta estrategia se ha aplicado en algoritmos de tipo Apriori para incrementar el rendimiento del algoritmo de cálculo de itemsets frecuentes.

Estas estrategias utilizan tablas hash para obtener información de la BD y así poder eliminar ítems de las transacciones y hasta transacciones completas que se sabe que no van a aportar más al cálculo del soporte de los itemsets candidatos. Estas heurísticas se basan en las propiedades de Recorte de Transacción y Poda de Transacción. Estos plantean lo siguiente:

Un ítem de una transacción t puede ser eliminado de t (Recorte de Transacción) si:

- Este no aparece en al menos k de los k-itemsets candidatos contenidos en esa transacción.
- Este no aparece en al menos un (k+1)-itemset elegible de esa transacción, considerando como elegible aquel con todos sus k+1 itemsets de tamaño k contenidos en Ck (conjunto de candidatos de k-itemsets).

Si una transacción t puede contener a un (k+1)-itemset, entonces esta debe contener a (k+1) itemsets de tamaño k. En caso contrario esta puede ser eliminada (Poda de Transacción).

El método representativo de este tipo de algoritmos es el DHP [10] (Direct Hashing and Pruning), que se apoya en la reducción efectiva del tamaño de las transacciones y la disminución del número de transacciones a procesar para mejorar la eficiencia del los métodos de tipo Apriori.

Cuando este algoritmo recorre el conjunto de datos para calcular el soporte de los candidatos de nivel k o k-itemsets, genera información sobre cada (k+1)-itemset elegible como candidato, generando todas las combinaciones posibles de cada transacción y acumulándolo en una tabla hash. Después de esto, quedan en cada celda de la tabla hash la suma de la cantidad todos los itemsets de igual valor de hash de todas las transacciones. Este valor de celda representa una cota superior de los soportes de los itemsets asociados, por lo que los itemsets que se tendrán en cuenta serán lo que tengan ese valor mayor que el soporte mínimo establecido. Utilizando esta información se poda la transacción según las propiedades de Recorte de Transacción y Poda de Transacción.

El seudocódigo del algoritmo utiliza las siguientes notaciones: Ck es el conjunto de candidatos de k-itemsets, Hk es la tabla de hash para contar las ocurrencias de los (k+1)-itemsets en las transacciones.

Algoritmo: DHP
```
// Part 1
forall transaction t_i ∈ D do begin
  "Insert and count 1-itemset occurrences in a hash tree";
  forall 2-itemset x of t_i do
    H_2[h_2(x)]++ ;
end
L_1 = { c | c.Support ≥ minsup ∧
        "c exist in a leaf node of the hash tree" };
// Part 2
k = 2 ;
D_k = D ; // Database for large k-itemset
      // Make a hash table
while |{ x | H_k[h_k(x)] ≥ minsup }| ≥ LARGE do begin
  GenCandidate(L_k-1, H_k, C_k);
  D_k+1 = ∅;
  forall transaction t ∈ D_k do begin
    CountSupport(t, C_k, k, t');     // t' ⊆ t
    if |t'| > k then begin
      MakeHashT(t', H_k, k, H_k+1, t'');
      if |t''| > k then D_k+1 = D_k+1 ∪ {t''};
    end
  end
  L_k = { c ∈ C_k | c.Support ≥ minsup };
  k++;
end
// Part 3
GenCandidate(L_k-1, H_k, C_k);
while |C_k| > 0 do begin
```

$D_{k+1} = \varnothing$;
forall transaction $t \in D_k$ **do begin**
CountSupport(t, C_k, k, t'); // $t' \subseteq t$
if $|t'| > k$ **then** $D_{k+1} = D_{k+1} \cup \{t'\}$;
end
$L_k = \{ c \in C_k \mid c.\text{Support} \geq minsup \}$;
if $|D_{k+1}| = 0$ **then break**;
C_{k+1} = apriori_gen(L_k); // refer to Apriori
k++;
end
Answer = $\bigcup_k L_k$;

Fig. 3. Seudocódigo del algoritmo DHP.

Procedimiento: GenCandidate
Entrada: L_{k-1} – Conjunto de (k-1)-itemsets frecuentes
H_k – Tabla de hash de los k-itemsets posibles candidatos
Salida: C_k – Conjunto de k-itemsets candidatos

$C_k = \varnothing$;
foreach itemset $c = \{c_p[1], c_p[2], \ldots, c_p[k\text{-}2], c_p[k\text{-}1], c_q[k\text{-}1]\}$
and $c_p, c_q \in L_{k-1}$ **and** $|c_p \cap c_q| = k\text{-}2$ **do**
if $H_k[\mathrm{h_k}(c)] \geq minsup$ **then**
$C_k = C_k \cup \{c\}$; // Insert c into a hash tree

Fig. 4. Seudocódigo del procedimiento GenCandidate.

Procedimiento: CountSupport
Entrada: t – Transacción que está siendo analizada
C_k – Conjunto de k-itemsets candidatos
k – Tamaño de los itemsets candidatos
Salida: t' – Transacción que se analiza modificada

// Count support and make transaction trimming
forall itemset $c \in C_k$ **and** $c = \{t_{i_1}, t_{i_2}, \cdots, t_{i_k}\} \subset t$ **do begin**
c.Support ++;
for ($j = 1$; $j \leq k$; j++) **do** $a[i_j]$++;
end
for ($i = 0$; $i < |t|$; i++) **do**
if $a[i] \geq k$ **then** $t' = t' \cup \{t_i\}$;

Fig. 5. Seudocódigo del procedimiento CountSupport.

Procedimiento: MakeHashT
Entrada: t' – Transacción que será evaluada

H_k – Tabla de hash de los k-itemsets
k – Tamaño de los itemsets de la tabla H_k
Salida: t'' – Transacción t' modificada
H_{k+1} – Tabla de hash de los (k+1)-itemsets

// Make hash table and a further transaction trimming
forall (k+1)-itemset $x = \{t'_{i_1}, t'_{i_2}, \cdots, t'_{i_{k+1}}\} \subset t'$ **do**
 if "$\forall$ k-itemset $y \subset x$, $H_k[h_k(y)] \geq minsup$" **then begin**
  $H_{k+1}[h_{k+1}(x)]$++ ;
  for ($j = 1$; $j \leq k+1$; j++) **do** $a[i_j]$++ ;
 end
for ($i = 0$; $i < |t|$; i++) **do**
 if $a[i] > 0$ **then** $t'' = t'' \cup \{t'_i\}$;

Fig. 6. Seudocódigo del procedimiento MakeHashT.

Es importante notar que la eficiencia de la poda y recorte de transacciones aumenta en correspondencia con el tamaño de la tabla de hash, ya que al ser estas más grandes se reducen las colisiones y se incrementa la cantidad de podas y recortes.

3.3 Particionamiento

Los métodos basados en particionamientos de las transacciones se propusieron con el objetivo de explotar la ventaja de mantener la base de datos en memoria y evitar las operaciones de E/S en disco. El método representativo de estos algoritmos es el Partition [11], el cual divide las transacciones de la base de datos en tantas partes disjuntas (particiones no necesariamente de iguales tamaños) como fueran necesarias para que todas las transacciones en cada partición pudieran almacenarse en la memoria operativa.

Este algoritmo recorre la base de datos en dos fases. En la primera, cada partición es minada independientemente para encontrar todos los itemsets frecuentes localmente. Al final de esta fase se toman los itemsets localmente frecuentes como itemsets candidatos globalmente. En la segunda fase, se determinan los soportes de los itemsets candidatos y se identifican los itemsets frecuentes.

Este particionamiento se sustenta en la propiedad de particionamiento del soporte de un itemset. Esta plantea que si un itemset no es localmente frecuente en ninguna parte, o subconjunto, de una partición de una base de datos, entonces no será frecuente globalmente. De esta propiedad de infiere que los itemset que no son frecuentes localmente no serán frecuentes globalmente, por lo que se excluyen de los candidatos globales.

En este algoritmo se obtienen los conjuntos frecuentes de cada partición utilizando un algoritmo de tipo apriori (GenItemsets). En el seudocódigo se considera una partición P de m subconjuntos p1 a pm.

Algoritmo: Partition

for ($i = 1$; $i \leq m$; i++) **do begin** // Phase I
 "Load partition $p_i \in P$";

```
  GenItemsets(p_i, L_i);
end
C = ∪_{i=1..m} L_i ;
for ( i = 1; i ≤ m ; i++ ) do begin // Phase II
  "Load partition p_i ∈ P";
  AccumulateSupport(p_i, C);
end
L = {c ∈ C | c.Support ≥ minsup};
Answer = L
```

Fig. 7. Seudocódigo del algoritmo Partition.

3.4 Estructuras Arbóreas

Entre los métodos descendentes con recorrido en profundidad se encuentran como principal grupo los métodos basados en arboles. Estos métodos tienen la ventaja que realizan una representación compacta de la base de datos, permitiendo el cálculo del soporte de manera eficiente y requiriendo solamente dos pasadas por la base de datos. En vez de generar un gran número de candidatos, se conserva (de forma compacta) el grupo esencial de elementos originales de la base de datos. Debido a esto, el análisis se centra en el cálculo del soporte de itemsets y no en generar candidatos. En vez de recorrer toda la base de datos para verificar si un itemset es frecuente, se particionan los itemsets para que sean examinados sólo cuando sea necesario. Aquí se pone de manifiesto la estrategia de divide y vencerás, reduciendo sustancialmente el espacio de búsqueda y elevando la eficiencia del algoritmo.

El algoritmo representativo de este grupo es el FP-Growth [12]. Este algoritmo que se basa en el crecimiento de ítems o patrones frecuentes, utilizando una estructura compacta y eficiente de datos denominada FP-Tree.

El FP-Tree es una estructura que almacena información importante acerca de los conjuntos frecuentes que se encuentran en la base de datos. Solo los ítems frecuentes tendrán asociados nodos en el árbol; a su vez, los nodos en el árbol son reordenados de forma tal que los de mayor soporte tengan más probabilidad de compartir nodos que otros con menos soporte. Esta estructura (compacta) de datos es diseñada basada en las siguientes consideraciones:

- Como los itemsets frecuentes son los que resultan relevantes en el proceso de búsqueda de ítems frecuentes, es necesario realizar una iteración sobre la base de datos para determinar cuáles son estos ítems.
- Si se almacena en alguna estructura compacta el conjunto de ítems frecuentes de cada transacción, entonces se evita tener que hacer iteraciones reiteradas e innecesarias sobre la base de datos.
- Si múltiples transacciones tienen el mismo conjunto de ítems frecuentes, entonces estas pueden unirse en una sola que almacene el número de veces que ocurre esto. Esta operación se puede realizar de forma sencilla si los ítems en cada transacción estuvieran ordenados de acuerdo a un orden prefijado.

- Si dos transacciones comparten un prefijo común, de acuerdo a un cierto orden prefijado, entonces las partes comunes pueden unirse en una sola estructura de prefijos que almacene también el número de veces que ha ocurrido esto. Si los ítems de cada transacción son ordenados en orden descendente de acuerdo a su soporte, entonces existirá mayor probabilidad de que se compartan mas prefijos y con ello se reduzca el gasto de memoria de almacenaje y a su vez se eliminen iteraciones innecesarios sobre la base de datos.

Con estas observaciones se puede construir el FP-Tree a través de los siguientes pasos:

1. Se realiza una iteración sobre la base de datos para determinar los ítems frecuentes. Luego, estos son ordenados descendentemente de acuerdo a su soporte e insertados en una lista *L.*
2. Se crea la raíz del FP-Tree y se inicializa como vacía. Posteriormente, para cada transacción *t* de la base de datos se realizan las siguientes acciones:
   - Se seleccionan de *t* los ítems que resultaron frecuentes y se insertan en la lista *P* ordenadamente de acuerdo al orden que ocupan en la lista *L.*
   - Esta lista de ítems frecuentes se le pasa como parámetro a un procedimiento que se encarga de adicionar una transacción al FP-Tree siguiendo las consideraciones planteadas anteriormente.

El árbol del FP-Tree posee las siguientes características:

- La cantidad de nodos hojas coincide con la cantidad de conjuntos de ítems únicos diferentes entre todas las transacciones de la base de datos, estando la cantidad de los nodos intermedios en correspondencia con la cantidad de prefijos coincidentes entre todas las transacciones.
- La altura del árbol depende de la cardinalidad del conjunto de ítems únicos mayor entre todas las transacciones.
- Para cada ítem frecuente ai, todos los posibles itemsets frecuentes en los que puede estar contenido ese ítem se pueden encontrar navegando a través de la lista enlazada que contiene a los nodos que almacenan al ítem frecuente ai y que se puede acceder a partir de la correspondiente celda del Header Table.

El algoritmo FP-Growth, el cual consta de los siguientes pasos:

1. Construir la base condicional del patrón frecuente para cada patrón en el *Header Table.*
2. Construir el FP-Tree condicional de cada base condicional del patrón frecuente analizado.
3. Realizar el minado recursivo del FP-Tree condicional y repetir el proceso sobre los patrones frecuente obtenidos. Si el FP-Tree condicional contiene un solo camino, entonces todas las combinaciones de patrones que se pueden formar con los ítems que representa cada nodo son frecuentes.

La idea general seguida para efectuar el proceso de minado es la siguiente:

1. Para cada patrón frecuente, se construye su base condicional, y luego su FP-Tree condicional.
2. Repetir el proceso en cada FP-Tree condicional creado hasta que el FP-Tree resultante sea nulo, o contenga sólo un camino.

4. Computo reconfigurable

La ley de Moore, que establece que la tecnología de los semiconductores avanza de manera exponencial, ha mantenido su validez por más de tres décadas. Los expertos predicen que seguirá así durante otro decenio. Cuando se introdujeron por primera vez los circuitos integrados, los paquetes lógicos contenían una docena o una cantidad similar de transistores. En la actualidad, debido a incrementos exponenciales de la densidad de los circuitos, los chips de microprocesadores han rebasado la barrera de los 10 millones de transistores.
Para poder hacer frente a las capacidades de los nuevos componentes electrónicos, las técnicas de diseño han cambiado drásticamente. De la fabricación a mano de los circuitos lógicos en sus inicios hasta el desarrollo de circuitos a partir de descripciones de alto nivel. [13]

La configuración reconfigurable o computo reconfigurable es un campo de investigación relativamente nuevo. No fue hasta mediados de los 80 en que se empezaron a ver los primeros avances significativos en el área. Esta surge con el objetivo de dotar a sistemas con la potencia de cálculo que presentan los circuitos de aplicación específica y de la flexibilidad que presenta el software para resolver problemas. El cómputo reconfigurable se encuentra en un estado intermedio entre estos dos paradigmas de diseño.

4.1 Conceptos Básicos

Antes de la aparición del cómputo reconfigurable se abordaban dos enfoques para el diseño y ejecución de algoritmos. El primero se refiere al uso de los ASICs (Application Specific Integrated Circuits). Estos son dispositivos electrónicos los cuales se diseñan con una funcionalidad específica y bien definida. Estos dispositivos tienen la ventaja que al ser fabricados desde un principio con un objetivo, los circuitos que lo componen se pueden optimizar de manera muy eficiente en cuanto al área que ocupan, el consumo energético o la frecuencia de operación. Estos logran ser muy eficientes en su desempeño en estos tres aspectos. También al fabricar estos por lotes muy grandes, su costo es considerablemente bajo.

Estos circuitos tienen la desventajas que sus ciclos de diseño son muy largos, ya que estos hay que diseñarlos, mandar a hacer prototipos y probarlos. Si algo en el proceso de diseño no quedo como se esperaba, hay que empezar el ciclo otra vez. La dificultad que tiene esto es que no siempre el proceso de diseño lo ejecuta el que fabrica el ASIC, por lo que la creación del prototipo y rectificación se pueden tornar procesos tediosos y largos. Por otra parte presentan cerro flexibilidad en cuanto a modificación una vez fabricados, por lo que si se detectan errores una vez fabricados en masa, se pierde gran cantidad de recursos.

El segundo enfoque es el referido al diseño de software que se ejecutan sobre plataformas hardware de propósito general. Esto es lo referente a la ejecución de instrucciones por microprocesadores. Este tipo de enfoque da la posibilidad de una gran flexibilidad en cuanto a su modificación, ya sea para corregir errores o agregar o modificar funcionalidades, siendo su costo casi nulo. Otra característica es que los ciclos de diseño de este tipo de sistemas son muy cortos y en general, se logran obtener aplicaciones muy grandes y con controles intensivos de los flujos de datos en muy poco tiempo. Todas estas ventajas se obtienen a costa de la potencia de cálculo. Este tipo de enfoque tiene la desventaja que las velocidades de ejecución y el consumo de potencia son mucho más elevados que los ASICs. Esto es debido a que como se ejecutan en microprocesadores

de propósito general, se tienen que hacer las funciones complejas mediante la ejecución de muchas funciones simples y generales, ralentizando el proceso e introduciendo overhead en la ejecución de cada instrucción, ya que estas tienen que ser leídas de la memoria y ser decodificadas para poder ser ejecutada.

El cómputo reconfigurable se encuentra en un estado intermedio entre los dos enfoques anteriores. Mientras se obtienen la flexibilidad de los diseños de software, en el computo reconfigurable se logran tiempos de ejecución y consumo de potencia comparables con los ASICs, mejorando considerablemente los tiempos de los ciclos de diseño, haciéndolos mucho más sencillos y menos demorados. Por otra parte se puede destacar que estos tienen costos superiores a los ASICs pero no son tan elevados como los de los microprocesadores de propósito general. El enfoque del cómputo reconfigurable se usa principalmente para el prototipado de circuitos que serán implementados posteriormente en ASICs y como aceleradores de aplicaciones, ejecutando algoritmos en su totalidad o como coprocesadores en diseños híbridos hardware-software.

Este tipo de enfoque se vio potenciado en la década del 80 con la introducción de los FPGAs (Field-Programmable Gate Arrays). Estos dispositivos presentan gran flexibilidad ya que permiten ser reconfigurados, cambiando su funcionalidad parcial o completamente. Estos dispositivos tienen herramientas que permiten hacer los diseños de los circuitos lógicos con lenguajes de descripción de hardware de alto nivel, encapsulando al usuario de muchos aspectos que se resuelven automáticamente. Esto permite reducir considerablemente los ciclos de diseño, reduciendo el costo final del producto.

Los diseños que utilizan FPGAs tienen una ventaja importante sobre los diseños software y es que presentan niveles de aceleración elevados respecto a estos últimos. En las aplicaciones que utilizan FPGAs se reportan resultados desde 10 y hasta 1000 veces más rápidos. Esta aceleración se hace mayor en algoritmos que presentan alto grado de paralelismo, donde se puede llevar este a nivel de instrucción y tienen cálculos intensivos sobre los datos.

Las primeras aplicaciones del computo reconfigurable estuvieron orientadas al procesamiento de señales e imágenes, aunque en la actualidad se extienden a algoritmos de cifrado, equipos de radio implementados por software, sistemas aeroespaciales y de defensa, prototipos de ASICs, sistemas de visión por computadoras, reconocimiento de voz, bioinformática, emulación de hardware mediante computadora, entre otras.

4.2 FPGAs

Los circuitos integrados se han convertido en elementos imprescindibles en la sociedad moderna. Toda una gama de ellos se encuentran presentes en las nuevas tecnologías de comunicaciones e informáticas, en los dispositivos móviles y computadoras personales. Los FPGAs son una alternativa sumamente atractiva para los desarrolladores e ingenieros por los costos relativamente bajos de estos y por las potencialidades que aporta su uso.

Los FPGAs son arreglos de compuertas lógicas programables. Las densidades de estas compuertas que estos dispositivos presentan hoy en día, permiten crear circuitos lógicos considerablemente grandes. La arquitectura de los FPGAs se basa en un gran número de pequeños bloques utilizados para reproducir sencillas operaciones lógicas que cuentan a su vez con flips-flops. Esto dota a estos dispositivos de una gran libertad de interconexión y por ende, una gran flexibilidad de diseño, permitiendo desde el desarrollo de funciones complejas para la construcción

de microprocesadores hasta la construcción de simples compuertas lógicas como los ANDs o los ORs.

Esta característica convierte a los FPGAs en una herramienta muy versátil, permitiendo su utilización en la construcción de circuitos a la medida sin los riesgos tecnológicos asociados a otras opciones de desarrollo. Esto es la causa principal por lo que los FPGAs se empleen en casi todas las aéreas donde se requiera de computación de alto rendimiento.

Las principales características de las FPGAs son la alta complejidad, el bajo costo de desarrollo, son fáciles de depurar, tolerancia a errores, pocas unidades, tamaño reducido, fiabilidad alta, una relación area-time-power intermedio y confidencialidad baja.

4.3 Flujo de diseño

Los sistemas reconfigurables regularmente están formados por un elemento de lógica reprogramable y un microprocesador de propósito general. El microprocesador realiza las tareas que no pueden ser realizadas eficientemente por la lógica reprogramable, además de programar y controlar el dispositivo reprogramable. El FPGA tiene la función de un coprocesador y realiza las tareas que al software le tomarían la mayor parte del tiempo, en especial, operaciones en donde no exista dependencia de datos, y así poder realizar varias acciones al mismo tiempo dentro del FPGA.

El flujo de diseño que se sigue para implementar algoritmos bajo este tipo de sistema de cómputo reconfigurable se muestra en la siguiente figura.

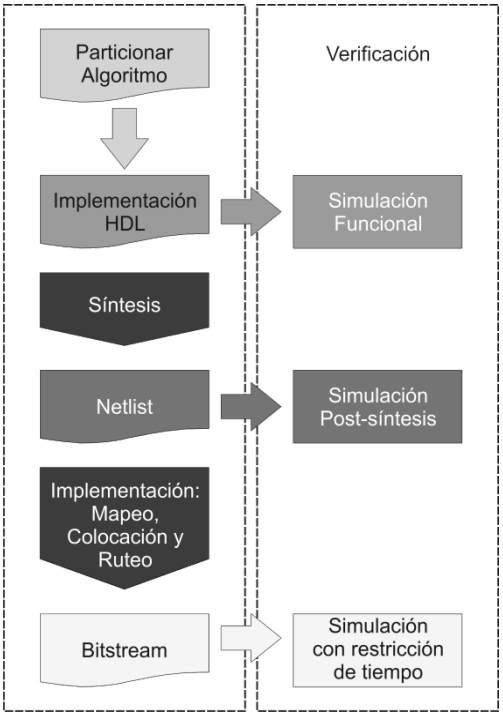

Fig. 8. Flujo de diseño típico en FPGAs.

El paso inicial consiste en particionar el algoritmo, para así identificar, a criterio del diseñador, aquellas partes que puedan ser implementadas eficientemente ya sea en hardware o en software. Para cada sección que se implemente en hardware, se crea un diseño utilizando los medios disponibles para ello, como lo pueden ser lenguajes de descripción de hardware (Hardware Description Languages, HDL) como VHDL, Verilog o System C por mencionar algunos; o también herramientas para el diseño de esquemáticos. Ambos medios describen el funcionamiento que el FPGA asumirá en tiempo de ejecución.

Posteriormente, estos diseños son sintetizados (systhesis phase) a nivel de compuertas y registros, con lo que se obtiene un netlist de elementos lógicos, aún independiente de la tecnología del FPGA a usar. La netlist es mapeada (Map phase) hacia los bloques lógicos del FPGA, es decir, el diseño lógico obtenido en la síntesis, es convertido a un diseño que involucra a la tecnología utilizada en los bloques lógicos del FPGA objetivo.

En la etapa de colocación (Place phase) se asignan las funciones lógicas en bloques particulares del FPGA. Enseguida, estos bloques son ruteados (Routing phase) por medio de las interconexiones programables para comunicar unos bloques con otros de acuerdo al diseño planteado. Al final del flujo de diseño, se obtiene el archivo de programación (bitstream) con el cual se configurará el FPGA.

En cualquiera de sus etapas, el flujo de diseño puede realizarse de forma manual o automática. La forma automática requiere un mínimo esfuerzo del programador para la creación de los diseños, además no necesita tener conocimiento acerca de la tecnología del FPGA donde se implementará el diseño. Sin embargo, si se comparara contra una implementación manual, la implementación automática podría tener un menor rendimiento en la ejecución del algoritmo, y el diseño resultante podría ocupar una mayor cantidad de los recursos disponibles del FPGA.

La forma manual resulta en una implementación mucho más eficiente en uso de recursos y rendimiento, pero se requiere que el programador conozca muy bien la tecnología y estructura interna del FPGA objetivo. Además, constantemente se deben realizar procesos de simulación y depuración, no solo para asegurar el correcto funcionamiento del circuito, sino también para prevenir posibles conexiones equivocadas que puedan causar cortos dentro del FPGA, y en consecuencia dañarlo [14].

5. Implementación de métodos de obtención de conjuntos frecuentes sobre FPGAs

Pocas investigaciones se han llevado a cabo para realizar implementaciones eficientes sobre hardware de estos tipos de algoritmos. Esto puede ser debido al reto que representa implementar arquitecturas y funciones de pertenencia entre conjuntos de ítems eficientes, así como la complejidad que aporta el control en estas aplicaciones que tienen como característica el manejo de grandes volúmenes de datos.

5.1 Apriori

Uno de los diseños del Apriori para la implementación hardware[2] sobre un FPGA Xilinx Virtex-II Pro 100 con 44,000 slices mejora su eficiencia como mínimo 4 veces. Este diseño tiene como elemento clave el uso de una arquitectura de arreglo sistólico con unidades de procesamiento conectadas como un arreglo lineal. Cada unidad contiene memorias para almacenar el candidato que se le está calculando el soporte y para permitir las esperas en el arreglo, un contador de índice y un comparador, que permite comparar la salida de la memoria de candidatos con el dato de entrada (t). Como todos los conjuntos de ítems (ya sean los candidatos generados o las transacciones de la base de datos) tienen sus elementos ordenados lexicográficamente solo hace falta las señales de igual o mayor que para efectuar las operaciones de pertenencia de conjuntos.

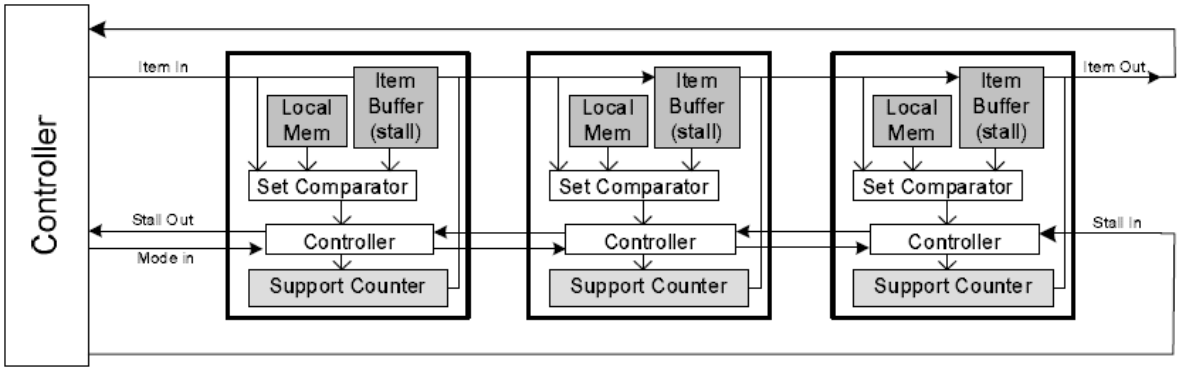

Fig. 9. Estructura del arreglo sistólico.

La estrategia en esta implementación es hacer circular los datos de la misma forma que circula en un pipeline. Los datos entran por un extremo del arreglo lineal, desde el bloque controlador, por los diferentes canales que tiene (ítems, control, información de demoras), representando, tanto los itemsets como las transacciones de la base de datos, en forma de lista horizontal de ítems.

Para la etapa del cálculo de soporte se hace circular por el arreglo (ítem a ítem) los k-itemsets candidatos para cargar las unidades de procesamiento. Después que el primer itemset candidato esté almacenado en la primera unidad, el i-ésimo itemset candidato es reenviado hacia la i-ésima unidad, y así hasta que el arreglo este lleno. Esto requiere de $k * |C_k|$ ciclos de reloj. En caso de que la cantidad de unidades en el arreglo sea menor que la cantidad de k-itemset candidatos ($c_a < |C_k|$) este proceso se tendría que repetir hasta que se acabaran los itemsets ($|C_k|/c_a$).

Después que las unidades están cargadas con los itemsets candidatos se crea un flujo de datos hacia el arreglo con las transacciones de la base de datos, un ítem en cada ciclo y se va comparando con el itemset almacenado previamente, si al terminar la transacción el itemset candidato se encontraba en esta, se aumenta el contador. Al terminar este proceso, se requiere obtener la información del conteo de soporte que se encuentra en cada unidad.

Para la etapa de generación de candidatos se utiliza la misma estrategia que para el cálculo de soporte. Se circulan los k-1itemsets frecuentes para cargar las unidades de procesamiento, lo que para este propósito se crea el flujo de datos con los mismos k-1 itemsets frecuentes. En este proceso (al igual que el cálculo de soporte) se tiene que comparar los itemsets para generar los candidatos de nivel k, y al haber una coincidencia en los primeros k-2 ítems de los itemset (el almacenado en la

unidad y el que está circulando por el arreglo sistólico) se hace la combinación de estos y se genera el nuevo candidato. Al coincidir, si el ítem k-1 que se encuentra en la memoria de la unidad es mayor que el que está en el flujo de datos, se inyecta este ítem en el flujo (al final del itemset que causo la coincidencia). Cada vez que es inyectado un elemento en el datapath, es inyectada también una demora hacia las unidades de procesamiento anteriores para que no se genere conflicto en el flujo de datos. Cuando este flujo sale del arreglo están los itemsets candidatos (los primeros k-1 ítems) con la lista de sufijos que componen (cantidad de demoras inyectadas al arreglo).

Para la etapa de poda se necesitan comparar los k-itemsets candidatos con los k-1 itemsets frecuentes para comprobar que todos los subconjuntos de tamaño k-1 de los k-itemsets candidatos son frecuentes. Para esto se hace un flujo con los candidatos y se inyecta en el flujo información de cotejo entre estos ítems. Como esta información llega al control en cuanto sale del arreglo le es fácil determinar que ítems no estaban presentes en los candidatos y así determinar cual cumple con la condición de poda y no pasa a la próxima etapa.

Otro aspecto significativo en esta implementación hardware fue el sistema de demora del arreglo sistólico para poder inyectar datos en el flujo que circulaba por él en las etapas de generación de soporte y poda, ya que estos se generaban de forma impredecible. Como cada unidad de procesamiento envía el flujo de las transacciones además de que puede generar aleatoriamente algún dato necesario para el control del sistema, es necesario retrasar en un ciclo todo el procesamiento de las unidades anteriores cada vez que esto pase, evitando que se pierdan datos del flujo o que se generen datos nuevos y se cree un conflicto. Para esto se interconecto la salida stall de cada unidad de procesamiento a la entrada de stall de la unidad anterior, propagándose en orden inverso al flujo de datos. Además se equipó a cada unidad con una memoria de demora en caso de que en alguna unidad anterior se requiera inyectar un elemento a la misma vez y no se pierda.

Debido a la falta de implementaciones de hardware de estos algoritmos, los autores hicieron las pruebas con la mejor implementación software reportada (APRIORI BRAVE, Borgelt y Goethals). Se utilizaron las bases de datos T40I10D100K (15 MB) y T10I4D100K (4 MB) y se realizaron pruebas con varios soportes. Los algoritmos se ejecutaron en una computadora dual Xeon a 2.8Ghz y 3 GB de memoria RAM. La arquitectura la implementaron para el dispositivo Virtex II Pro XC2VP100 con -6 en el grado de velocidad, proveyendo a la arquitectura un ancho de banda entre el arreglo sistólico y la memoria de 250 MB/s. Bajo estas condiciones, la mínima aceleración lograda fue de 4x.

Otra implementación del Apriori [3] parte de la arquitectura anteriormente explicada e introduce un nuevo elemento para aprovechar la redundancia que existe entre los elementos candidatos a ser procesados, ya que entre k-itemsets adyacentes, por lo general hay pocos ítems de diferencia. Este diseño se centra en acelerar el cálculo de soporte ya que la determinación de la pertenencia o no de un itemset a una transacción es el cuello de botella en el algoritmo.

Con este objetivo los autores introducen el uso de una CAM (Content-Addressable Memory) asociada a una memoria de mapa de bits (bitmap memory) de la siguiente forma:

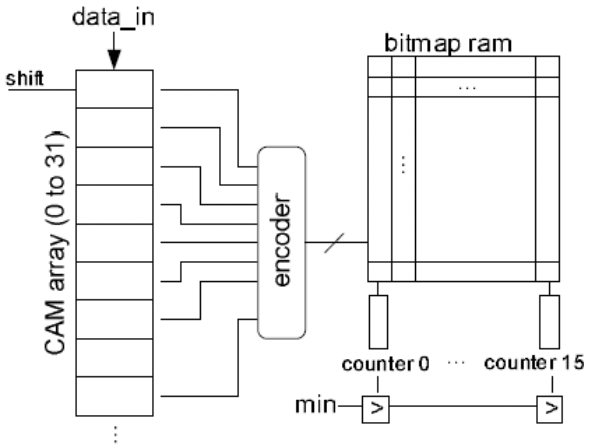

Fig. 10. Estructura del Bitmapped CAM

Esto permite a cada bloque manejar los itemsets que se puedan representar por 32 ítems continuos y hasta un máximo de 16 candidatos (este número puede variar según la implementación). La CAM (S^b_{cam}) almacena los itemsets candidatos de la siguiente manera:

$b = 0$
$\forall c \in C_k \; do$
  $if\left(\left|S^b_{cam}\right| + \left|c \backslash S^b_{cam}\right| > 32\right)$
    $b++$
  $else$
    $S^b_{cam} = S^b_{cam} \cup c$

Por otra parte, todos los candidatos de ese bloque son representados en el mapa de bits y es llenado de la siguiente forma:

$\forall i \in S^b_{cam} \; do$
  $\forall c \in C_k \; do$
    $if \; i \in C_i$
      $bitmap_b(i,c) = 1$
    $else$
      $bitmap_b(i,c) = 0$

Cuando una transacción es pasada al arreglo sistólico, cada ítem se presenta a la memoria CAM y esta devuelve una dirección que corresponde con este ítem en el mapa de bits, incrementando el contador de cada candidato que contenga ese ítem ($\mathrm{cont_i} \mathrel{+}= \mathrm{bitmap_b(i,c)}$). Si al final de la transacción el contador es igual a k, entonces el candidato es un subconjunto de la transacción. En todos los casos al final de cada transacción, el contador de ítems del candidato es reiniciado y son incrementados los contadores de soporte asociados a los candidatos que se encontraron en la transacción. Este proceso requiere solo de $\left\lceil \frac{\mathrm{c_k}}{\mathrm{c_a}} \right\rceil (\mathrm{t}+1)$ ciclos donde t es la cantidad de ítems en la base de datos y $\mathrm{c_a}$ la cantidad de bloques de 16 capacidades de candidatos que se puedan poner en el arreglo sistólico (limitados por el área).

Los autores hicieron las pruebas con la mejor implementación software reportada (APRIORI BRAVE, Borgelt y Goethals). Se utilizaron las bases de datos T40I10D100K (15 MB) y T10I4D100K (4 MB) y se realizaron pruebas con varios soportes. Los algoritmos se ejecutaron en una computadora dual Xeon a 2.8Ghz y 3 GB de memoria RAM. La arquitectura la implementaron para el dispositivo Virtex II XC2V6000 con -4 en el grado de velocidad, en una plataforma de hardware reconfigurable SRC-6 MAPstation. Bajo estas condiciones, la mínima aceleración lograda fue de 24x.

5.2 DHP

En esta implementación hardware del algoritmo DHP[4] los autores proponen una arquitectura basada en tablas de hash y pipelines a la que denominaron HAPPI (Hash-based and PiPelIned). Esta arquitectura está basada en la implementación hardware del algoritmo apriori de Baker y Prasanna. Específicamente utilizan el arreglo sistólico propuesto por los autores y le incorporan un filtro para la poda (Trimming Filter) y la tabla de hash (Hash Table Filter) para obtener la información de poda.

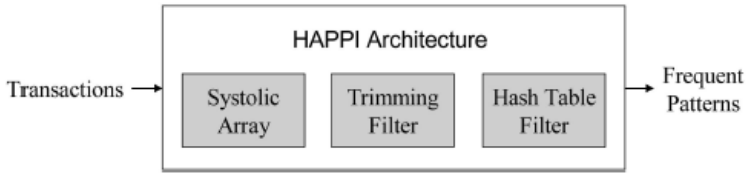

Fig. 11. Arquitectura HAPPI [4].

Esta arquitectura está compuesta de tres módulos. Primero, la base de datos es alimentada en la arquitectura y los itemsets candidatos son comparados con las transacciones en el arreglo sistólico y se obtienen los conjuntos frecuentes de nivel k. Al mismo tiempo, se determina en la misma unidad del procesamiento del arreglo sistólico la frecuencia de aparición en las transacciones de cada ítem. Esta información se utiliza para eliminar los ítems que no son frecuentes ya que no aportan información útil. Esto se calcula para el filtro de poda. Después se generan todos los itemsets posibles de cada transacción y se insertan en la tabla de hash para irla construyendo. La información obtenida de esta tabla se utiliza para filtrar los candidatos. Después que se comparan los candidatos con los ítems en la base de datos, la información de poda es recolectada y la tabla de hash construida.

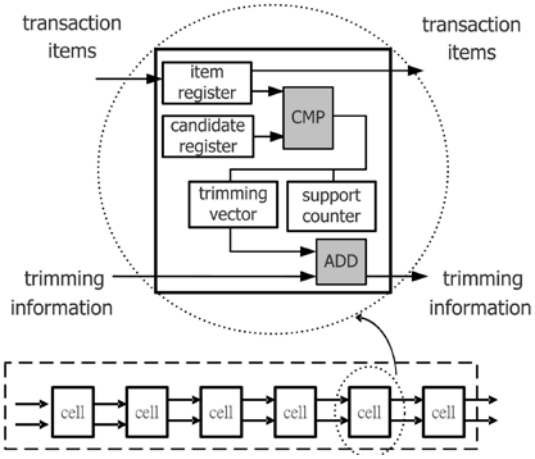

Fig. 12. Arreglo sistólico de la arquitectura HAPPI [4].

Al filtro de poda se le entra las frecuencias de aparición de los ítems (información de poda) que arroja el arreglo sistólico y los ítems de las transacciones, saliendo de este las transacciones ya podadas. Así mismo, al filtro de la tabla hash le entran las transacciones en el nivel k, y crea la tabla de hash con la información de poda para el siguiente nivel, y al mismo tiempo le están entrando los candidatos de nivel k y los está filtrando con la información generada en el nivel k-1.

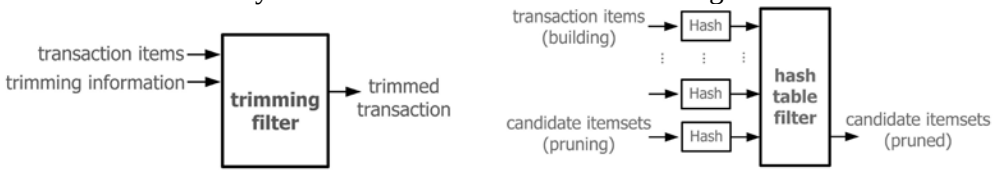

Fig. 13. Filtros de poda y tabla hash [4].

Los autores hicieron las pruebas con la implementación hardware propuesta en [2] y una implementación software del algoritmo DHP. Las arquitecturas se implementaron para el dispositivo Altera 1S40 FPGA con 50-MHz de frecuencia de reloj y una memoria SDRAM de 10 MB. Para este diseño lograron un arreglo sistólico de 500 celdas y 65536 posiciones en la tabla hash. Se utilizaron varias bases de datos generadas sintéticamente (T5I12, T10I4, T15I6, T20I8)D100K y se realizaron pruebas con varios soportes. Bajo estas condiciones, la mínima aceleración lograda fue de 47-122x con respecto a la arquitectura hardware propuesta por Baker y Prasanna, dando la mayor aceleración para la base de datos T5I2D100K. El algoritmo DHP se ejecutó en una computadora P4 a 3Ghz y 1 GB de memoria RAM. Los tiempos de ejecución del algoritmo en software son mucho menores que el de las arquitecturas hardware.

5.3 FP-Growth

Este diseño [5, 6] está basado en el algoritmo FP-growth, utilizando como punto clave en la arquitectura una estructura que los autores llaman Estructura de Árbol Sistólico.

Esta estructura consiste en un conjunto de elementos de procesamiento dispuestos en forma que se genere un "pipeline" a lo largo de una estructura arbórea multi-dimensional. En otras palabras, esta estructura es un conjunto de elementos de procesamiento (PE) que representan a los nodos de este árbol. Este árbol general esta implementado en forma de árbol binario, siendo el hijo izquierdo de cada nodo si primer hijo y el hijo derecho de cada nodo su siguiente hermano. Estas conexiones se utilizan para el pase de los elementos desde la raíz de este árbol (que sería la unidad de control por donde se inyectan los elementos).

En cada PE hay 2 banderas, match y InPath. Estas banderas se utilizan para guiar el recorrido que va a seguir un elemento por el árbol, mediante el pase del elemento entre los PE. Estas banderas se inicializan en 0 (match) y en 1 (InPath) respectivamente. Al entrar un elemento nuevo al árbol sistólico, este empieza a circular por todo el árbol sistólico.

Cada uno de los PE del árbol sistólico tiene 3 modos de trabajo: Write, Scan y Count. En el modo write es donde se inyectan los itemsets al árbol para construirlo. El modo Scan se utiliza para una vez construido el árbol, buscar las frecuencias de cada itemset.

Algoritmo: modo Write(ítem t)

Match ← 0; InPath ← 1

(1) If PE is empty
    do guarda ítem t
    count ← 1
    match ← 1
    stop forwarding
(2) If (t is in PE) and (InPath is 1)
    do match ← 1
    count ← count + 1
    stop forwarding
(3) If match is 0
    do forward t al siguiente hermano
    InPath ← 0
(4) else
    do forward t al primer hijo.

Algoritmo: modo Scan(ítem t)

bottom door ← open;

match ← 0; IsLeaf ← 0;

(1) If PE is empty
    stop forwarding
(2) If (t is in PE) and (bottom door is open)
    do match ← 1
    IsLeaf ← 1
    forward t al siguiente hermano
(3) If t < item in PE
    do IsLeaf ← 0

bottom door← close;
forward t al siguiente hermano
(4) If t > item in PE
do IsLeaf ← 0
forward t al siguiente hermano
forward t al primer hijo if bottom door is open

Algoritmo: modo Count(int CountRight, int CountBottom)

CountSent ←0;

CountChild ← CountRight + CountBottom;

If (CountSent = 0) and (IsLeaf = 1)
do CountSent ← 1;
forward(CountMySelf + CountChild) to Parent;
else
(CountChild) to Parent;

Al terminar de pasar todos los elementos de un itemset se restauran los valores de InPath y match a su valor de inicialización. Al terminar de inyectar todos los itemsets en el árbol sistólico se tiene en el cada camino del árbol, la coocurrencia de cada uno de los elementos que contiene dicho camino, y en el último PE del camino, la frecuencia con la que han coocurrido.

Esta estructura es dependiente de la aplicación y tiene que construirse teniendo en cuenta las características de los datos. Este árbol va a tener W niveles y K hijos de ramificación, obteniéndose una cantidad total de nodos de $\frac{K^{W+1}-1}{K-1}$.

6. Conclusiones

En este reporte se presento el problema del minado de conjuntos frecuentes así como las principales estrategias que existen para enfrentarlo. Se describió el marco teórico que sustenta a estas estrategias y se expusieron las implementaciones hardware de estos algoritmos que se ha reportado hasta la fecha.

Todos los diseños hardware expuestos en este trabajo utilizan como representación de los itemset y de la base de datos una lista horizontal de ítems. También se puede destacar que los diseños que mejores resultados presentan son los que menor cantidad de veces recorren la base de datos.

Un elemento característico de estas arquitecturas es que explotan los beneficios de los arreglos sistólicos, empleando elementos de procesamiento en donde el control del flujo de datos se logra principalmente por pequeñas estructuras de control en el interior de cada elemento de procesamiento y por la manera de interconexión de estos elementos, evitando el control centralizado del flujo de datos para que no se introduzcan líneas de comunicación largas y excesivas desde y hacia cada elemento de procesamiento.

Como trabajo futuro se propone explorar otras formas de representación de los itemsets y de la base de datos, específicamente la utilización de vectores de bits, dado la simplicidad que presentan las operaciones con estas representaciones, que por lo general están basadas en ANDs y ORs, las cuales son muy susceptibles a implementar en hardware y se sospecha que puedan tener altos niveles de aceleración. También se propone explorar técnicas de particionamiento de la base de datos para reducir la comunicación externa así como la valoración de co-diseños hardware software, ya sea implementar el software en una computadora, utilizando el FPGA como coprocesador o la utilización de kits de desarrollo que incorporen microprocesadores de propósito general. También se recomienda explorar técnicas de compactación para reducir el tamaño de la representación de la base de datos (sobre todo en bases de datos esparzas donde abundarían los 0s en una representación de vectores de bits).

Referencias bibliográficas

1. Agrawal, R. and R. Srikant. *Fast Algorithms for Mining Association Rules*. in *20th Int. Conf. Very Large Data Bases, {VLDB}*. 1994: Morgan Kaufmann.
2. Baker, Z.K. and V.K. Prasanna, *Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs.* 13' Anual IEEE Symposium on Field-Programmable Custom Computing Machine, 2005.
3. Baker, Z.K. and V.K. Prasanna, *An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems.* 14' Anual IEEE Symposium on Field-Programable Custom Computing Machine 2005, 2006.
4. Wen, Y.-H., J.-W. Huang, and M.-S. Chen, *Hardware-Enhanced Association Rule Mining with Hashing and Pipelining.* IEEE Transactions on knowledge and Data Engineering, 2008.
5. Sun, S. and J. Zambreno. *Mining Association Rules with Systolic Trees*. in *International Conference on Field-Programable Logic and it's Applications*. 2008.
6. Sun, S. and J. Zambreno. *A Reconfigurable Platform for Frequent Pattern Mining*. in *International Conference on ReConFigurable Computing and FPGAs*. 2008.
7. Agrawal, R., T. Imielinski, and A.N. Swami. *Mining Association Rules between Sets of Items in Large Databases*. in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*. 1993: ACM Press.
8. Zaki, M.J., et al., *New Algorithms for Fast Discovery of Association Rules*. 1997, The University of Rochester: New York.
9. Shenoy, P., et al. *Turbo-charging vertical mining of large databases*. 2000.
10. Park, J.S., M.-S. Chen, and P.S. Yu. *An Effective Hash Based Algorithm for Mining Association Rules*. in *Proc. ACM SIGMOD '95*. 1995.
11. Savasere, A., E. Omiecinski, and S.B. Navathe, *An Efficient Algorithm for Mining Association Rules in Large Databases.* 1995: p. 432--444.
12. Han, J., J. Pei, and Y. Yin. *Mining frequent patterns without candidate generation*. in *2000 ACM SIGMOD Intl. Conference on Management of Data*. 2000: ACM Press.
13. Wakerly, J.F., *Diseño digital, principios y prácticas*. 2001: Prentice Hall.
14. Chang, K.C., *Digital Systems Design with VHDL and Synthesis, An integrated approach*, I.C.S. Press, Editor. 1999.

RT_008, diciembre 2009
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