Capítulo 3

Reglas de asociación en bases de datos relacionales [*Mining Association Rules in Relational Databases*]

1. Introducción

La obtención de reglas de asociación se suele realizar a partir de grandes bases de datos transaccionales. En ellas puede ser interesante descubrir relaciones entre los distintos items presentes en las transacciones. Por ejemplo, la presencia de ciertos items en una que implica la presencia de otros items en la misma transacción. En las grandes bases de datos transaccionales, una transacción contiene la fecha en la que se produjo y los items involucrados en ella. Generalmente, cada registro de la base de datos incluirá un identificador de la transacción *TID* y uno de los artículos involucrados en ella.

Sin embargo, para aplicar los algoritmos de obtención de reglas de asociación a bases de datos relacionales, lo que podría resultar interesante para analizar comportamientos o realizar predicciones, debemos redefinir nuestro problema.

A partir de ahora, un ítem será un par de la forma $<atributo, valor>$ donde *atributo* hace referencia a la columna de la tabla de donde se deriva y *valor* es un identificador que corresponde a un valor o conjunto de valores del atributo. Se utilizarán la notación ***a:v*** para representar un ítem, donde *a* es un atributo y *v* es el valor que toma el atributo.

Una tupla t contiene a un ítem ***a:v*** si en su columna ***a*** se encuentra un valor ***v*** (en general, un valor perteneciente al dominio representado por ***v***). Una tupla *t* contiene a un conjunto de items [*itemset*] si contiene a cada uno de los items del conjunto. Un *k-itemset* es un conjunto de items con *k* elementos. Un itemset será relevante si aparece en una proporción de tuplas de la tabla superior a un umbral preestablecido al que denotaremos *MinSupport*.

Una regla de asociación será de la forma $X \Rightarrow Y$, donde X e Y son conjuntos de items de intersección vacía. La fiabilidad [*confidence*] de la regla de asociación $X \Rightarrow Y$ es la proporción de tuplas que, conteniendo a X, además contienen a Y. La relevancia [*support*] de la regla es la proporción de tuplas de la tabla que contienen tanto a X como a Y (contienen a $X \cup Y$). Una regla será de interés para nosotros si su fiabilidad alcanza un valor prefijado, al que llamaremos *MinConfidence*.

Una característica importante de los itemsets a partir de los cuales se obtienen las reglas de asociación es que sólo contendrán un ítem de cada atributo: si $a_1:v_1$ y $a_2:v_2$ pertenecen a un itemset entonces $a_1 \neq a_2$. Esto es consecuencia de la Primera Forma Normal [*1NF*]: una relación está en 1NF si y sólo si sus dominios simples contienen sólo valores atómicos.

EJEMPLO

Para ilustrar los distintos métodos mediante los cuales se pueden extraer reglas de asociación de una tabla se utilizará una sencilla tabla que podría ser una muestra de la base de datos de un banco:

Aval	Nómina	Crédito
No	No	No
No	No	Sí
No	Sí	Sí
Sí	Sí	Sí
Sí	Sí	Sí

Esta tabla podría provenir de la información recopilada por una entidad de crédito acerca de las solicitudes realizadas por sus clientes y de las decisiones tomadas en cada caso. Como es lógico, la tabla no recoge toda la información de la que debe disponer la entidad bancaria para decidir si se concede o no un crédito. No obstante, para determinados tipos de créditos (como los créditos consumo) puede que sea suficiente.

En esta tabla se recogen cinco casos de solicitudes de préstamos. En función del aval presentado por el cliente y de si éste dispone de una nómina (un trabajo fijo) se decide si se debe realizar el préstamo o no.

Lo lógico es que a una persona que tenga el aval requerido se le conceda inmediatamente el crédito (las dos últimas tuplas de la relación). Por otra parte, si el solicitante tiene domiciliada su nómina en el banco puede que se le conceda el crédito solicitado (podría tratarse de un pequeño crédito para la compra de algún electrodoméstico o algo así). Si además lleva años siendo cliente de la entidad quizá se le concedan créditos de mayor cuantía (como el crédito hipotecario para la adquisición de una vivienda).

Finalmente, si el cliente no dispone de aval alguno ni tiene ingresos fijos ningún banco le concederá un préstamo en principio. La concesión del crédito en el segundo de los casos recogidos en la tabla puede deberse a un error de mecanización (ruido en los datos de entrada), a que la tabla no recoja correctamente todos los factores involucrados en la concesión de créditos (obviamente no son sólo dos) o, simplemente, a tráfico de influencias (tan común en muchos sitios).

Para obtener las reglas de asociación derivadas de esta simple tabla se supondrá que *MinSupport* (la relevancia mínima de los itemsets relevantes) estará establecida en el *40%*: un itemset deberá aparecer, al menos, en dos de las cinco tuplas para ser calificado como relevante. El otro parámetro involucrado en la generación de reglas de asociación, *MinConfidence* (la fiabilidad mínima de una regla para que sea considerada de interés), será fijado en el *50%*.

Tras aplicar algún algoritmo de *Data Mining*, el usuario esperaría la obtención de un conjunto de reglas como el siguiente:

Nº	Regla de asociación	Fiabilidad
1	Aval:No ⇒ Nómina:No	2/3 66%
2	Nómina:No ⇒ Aval:No	2/2 100%
3	Aval:No ⇒ Crédito:Sí	2/3 66%
4	Crédito:Sí ⇒ Aval:No	2/4 50%
5	Aval:Sí ⇒ Nómina:Sí	2/2 100%
6	Nómina:Sí ⇒ Aval:Sí	2/3 66%
7	Aval:Sí ⇒ Crédito:Sí	2/2 100%
8	Crédito:Sí ⇒ Aval:Sí	2/4 50%
9	Nómina:Sí ⇒ Crédito:Sí	3/3 100%
10	Crédito:Sí ⇒ Nómina:Sí	3/4 75%
11	Aval:Sí ⇒ Nómina:Sí Crédito:Sí	2/2 100%
12	Aval:Sí Nómina:Sí ⇒ Crédito:Sí	2/2 100%
13	Aval:Sí Crédito:Sí ⇒ Nómina:Sí	2/2 100%
14	Nómina:Sí ⇒ Aval:Sí Crédito:Sí	2/3 66%
15	Nómina:Sí Crédito:Sí ⇒ Aval:Sí	2/3 66%
16	Crédito:Sí ⇒ Aval:Sí Nómina:Sí	2/4 50%

La fiabilidad de las reglas se expresa en la tabla como el cociente del número de casos en los que se verifica la regla de asociación partido por el número de casos en los que es cierto el antecedente de la regla y como porcentaje.

Como es lógico, de todas las reglas de asociación obtenidas sólo son interesantes algunas de ellas, como la séptima o la novena. Algunas de las reglas no son muy realistas debido al pequeño conjunto de datos de la tabla (por ejemplo, la tercera resulta claramente engañosa). Por otro lado, hay reglas redundantes: algunas reglas de asociación no aportan nada nuevo (como la duodécima respecto a la séptima).

2. Algoritmo t

Como primera adaptación de los algoritmos tradicionales de obtención de reglas de asociación (que suelen partir de bases de datos transaccionales) al caso particular de las bases de datos relacionales desarrollaremos a continuación un algoritmo derivado del *algoritmo Apriori*.

El *algoritmo Apriori* fue propuesto por Rakesh Agrawal y Ramakirshnan Skirant, dos investigadores del IBM Almaden Research Center, en su artículo "Fast Algorithms for Mining Association Rules" (1994). Este algoritmo supuso un avance considerable respecto a los algoritmos desarrollados anteriormente (como AIS o SETM) y sigue sirviendo de punto de referencia en todas las investigaciones actuales sobre el tema.

La adaptación propuesta aquí maneja única y exclusivamente tablas. No utiliza ninguna otra estructura de datos, por lo que podría programarse directamente en SQL. Éste es el motivo por el cual el algoritmo se denominará ***algoritmo t*** (*t* de *"tabla"*).

El *algoritmo t* permite obtener reglas de asociación directamente a partir una tabla (más concretamente, del conjunto de columnas de una tabla en el que estemos interesados). Los items de las reglas generadas son de la forma *a:v*, donde *a* es una columna de la tabla y *v* es un valor.

Los itemsets candidatos y los itemsets relevantes se almacenan en tablas independientes. Si los itemsets relevantes de mayor tamaño contienen N items, existirán $2*(N+1)+1$ tablas de itemsets como máximo: una para los items relevantes y un par de tablas para los k-itemsets candidatos y los k-itemsets relevantes (con $2 \leq k \leq N+1$). Lógicamente, la última tabla quedará vacía (la correspondiente a los itemsets de N+1 items). El número de columnas de la tabla sobre la que se aplica el algoritmo servirá de cota superior sobre el número de tablas necesarias: como es obvio, si tenemos C columnas en la tabla no nos podemos encontrar itemsets de más de C elementos.

El algoritmo aquí propuesto utiliza la técnica *"divide y vencerás"* para descomponer el problema tal como se hace en prácticamente la totalidad de los algoritmos de extracción de reglas de asociación ideados desde AIS (Agrawal, Imielinski & Swami: "Mining association rules between sets of items in large databases", ACM SIGMOD'93).

El proceso de obtención de las reglas de asociación, como es habitual, se descompone en dos etapas perfectamente diferenciadas:

① Obtener los itemsets relevantes

En esta fase se encuentran todos los itemsets con relevancia por encima del umbral preestablecido *MinSupport*; es decir, los itemsets relevantes. Recordemos que la relevancia de un itemset X es la proporción de tuplas que lo contienen, que denotaremos *support(X)*.

$$X \text{ es un itemset relevante si, y sólo si, } support(X) \geq MinSupport$$

② Generación de las reglas de asociación a partir de los itemsets relevantes

Una vez que se han obtenido los itemsets relevantes en la etapa anterior, la generación de las reglas de asociación es prácticamente directa. Si X y $X \cup Y$ son itemsets relevantes, la regla $X \Rightarrow Y$ tendrá una fiabilidad igual al cociente $support(X \cup Y)/support(X)$. La regla tendrá con seguridad una relevancia mínima (ya que $X \cup Y$ es un itemset relevante) y nos quedaremos con ella si su fiabilidad alcanza el umbral prefijado *MinConfidence*.

$$X \Rightarrow Y \text{ es una regla de asociación de interés si, y sólo si, } X \text{ es un itemset relevante,}$$
$$X \cup Y \text{ es otro itemset relevante} \quad y \quad support(X \cup Y)/support(X) \geq MinConfidence$$

2.1 Itemsets relevantes

Lo primero que debemos hacer para obtener las reglas de asociación derivadas de una tabla es extraer el conjunto de patrones que se repiten con cierta frecuencia, los itemsets relevantes de los que luego se derivarán las reglas de asociación.

Estos patrones se obtienen de una forma similar a la propuesta por Agrawal y Skirant en su algoritmo *Apriori*. Al igual que en cualquier otro algoritmo derivado de Apriori, se realizan múltiples pasadas sobre la base de datos para obtener los conjuntos de itemsets relevantes. En cada pasada se van obteniendo patrones de mayor longitud (itemsets con mayor número de elementos).

Inicialmente se obtienen los items individuales cuya relevancia alcanza el umbral mínimo preestablecido [*MinSupport*], con lo que se obtiene el conjunto *L[1]* de itemsets relevantes. En las siguientes iteraciones, se utiliza el último conjunto *L[k]* de k-itemsets relevantes obtenido para generar un conjunto de (k+1)-itemsets potencialmente relevantes (el conjunto de itemsets candidatos *C[k+1]*) y se obtiene la relevancia de estos candidatos para quedarnos sólo con aquéllos que son relevantes, que incluimos en el conjunto *L[k+1]*. Este proceso se repite hasta que no se encuentran más itemsets relevantes.

En los algoritmos AIS y SETM, los candidatos se generaban sobre la marcha, conforme se iban leyendo transacciones de la base de datos. El método utilizado implicaba la generación innecesaria de itemsets candidatos que nunca podían llegar a ser relevantes. En los algoritmos de la familia de *Apriori*, los candidatos se generan única y exclusivamente a partir del conjunto de itemsets relevantes encontrados en la iteración anterior.

Dado un itemset relevante, cualquier subconjunto suyo también es relevante. Por lo tanto, los k-itemsets candidatos del conjunto *C[k]* pueden generarse a partir de los itemsets del conjunto de (k-1)-itemsets relevantes *L[k-1]*. Además, se pueden eliminar de *C[k]* aquellos itemsets que incluyen algún itemset no relevante. Este proceso permite reducir el tamaño de los conjuntos de candidatos *C[k]* y mejorar de esta forma la eficiencia del algoritmo.

Al igual que en *Apriori*, en el algoritmo *t* se establece de antemano una ordenación entre los items de un itemset. En *Apriori*, los items de la base de datos se ordenaban lexicográficamente. En el algoritmo *t*, los items son de la forma *atributo:valor*. Dado que un itemset sólo puede contener un ítem de cada atributo de la relación (consecuencia de la atomicidad de los valores establecida por la Primera Forma Normal), se establece una ordenación entre los items de un itemset basada en un orden preestablecido entre los atributos de la relación (las columnas de la tabla de la base de datos sobre la que se aplica el algoritmo).

Como ya se ha comentado, el algoritmo ***t*** maneja únicamente tablas, por lo cual los conjuntos de itemsets relevantes *L[k]* y los conjuntos de itemsets candidatos *C[k]* se almacenan en tablas que contienen un par de columnas para identificar cada uno de los items del itemset (el par *atributo:valor*). En el caso de los itemsets relevantes, una columna adicional indica la relevancia del itemset (el número de tuplas en las que aparece).

El algoritmo ***t*** maneja la tabla de datos de entrada (tabla *Datos*), una tabla por cada conjunto de itemsets relevantes *L[k]* y una tabla por cada conjunto de candidatos *C[k]*. Las tablas *L[k]* contienen tuplas de la forma *(attr1,val1..attrK,valK,support)*. Por su parte, las tablas C[k] incluyen tuplas *(attr1,val1..attrK,valK)*.

El proceso de obtención de los itemsets relevantes de una tabla *Datos* utilizado por el algoritmo ***t*** se describe a continuación:

```
Obtener los items relevantes: L[1]

for (k=2; k≤Datos.columnas && L[k-1]!=∅; k++)

      Generar candidatos C[k]

      Obtener itemsets relevantes L[k]:
      L[k] = { (c,#c) | c∈C[k] ∧ #c ≥ MinSupport }

Itemsets relevantes: L = ∪ L[k]
```

Los itemsets relevantes tienen el número de columnas de la tabla de entrada como cota superior de su tamaño (al no poder un itemset contener más de un ítem de cada columna de la tabla), lo que justifica la condición $k \leq Datos.columnas$.

Así mismo, la generación de conjuntos de itemsets se puede detener en la iteración *k* cuando el conjunto *L[k-1]* no contenga ningún itemset: $L[k-1] \neq \varnothing$. En realidad, el proceso se podría detener en cuanto L[k-1] contuviese menos de k itemsets relevantes, ya que hacen falta dos itemsets de *L[k-1]* para generar un candidato de *C[k]* y k itemsets para que el candidato pueda llegar a formar parte de *L[k]*, siempre que cuente con la relevancia mínima:

```
Obtener los items relevantes: L[1]

for (k=2; k≤Datos.columnas && #L[k-1]≥k; k++)

      Generar candidatos C[k]

      Obtener itemsets relevantes L[k]:
      L[k] = { (c,#c) | c∈C[k] ∧ #c ≥ MinSupport }

Itemsets relevantes: L = ∪ L[k]
```

OBTENCIÓN DE LOS ITEMS RELEVANTES DE LA TABLA: *L[1]*

La pasada inicial del algoritmo debe determinar el conjunto de 1-itemsets relevantes a partir del cual realizar la generación del conjunto de candidatos *C[2]*. Este conjunto se puede obtener de una forma directa:

```
// C[1]: Todos los items de la tabla

for (i=1; i<=Datos.columnas; i++)

      INSERT INTO L[1]
      SELECT  i, Datos.columna(i), COUNT(*)
      FROM Datos
      GROUP BY Datos.columna(i)

// L[1]: Se eliminan los items no relevantes

DELETE FROM L[1]
WHERE support < MinSupport
```

La tabla L[1] contiene tuplas de la forma *(a,v,s)* donde *a* referencia a una columna de la tabla (que se supone están numeradas de *1* a *Datos.columnas*), v identifica un valor de esa columna y *s* es la relevancia del ítem *a:v* (expresada como el número de tuplas en las que aparece el valor *v* en la columna número *a*).

Ejemplo:

Mostremos el funcionamiento del algoritmo utilizando la tabla de ejemplo comentada anteriormente. Esta tabla contiene cinco tuplas:

Aval	Nómina	Crédito
No	No	No
No	No	Sí
No	Sí	Sí
Sí	Sí	Sí
Sí	Sí	Sí

Tabla de ejemplo

Para la tabla de ejemplo se introducirán inicialmente en la tabla *L[1]* (a la que denominamos *L1* por restricciones del lenguaje de definición de datos de Oracle) los seis items existentes en la relación con sus respectivas relevancias: *(Aval,No,3)*, *(Aval,Sí,2)*, *(Nómina,No,2)*, *(Nómina,Sí,3)*, *(Crédito,No,1)* y *(Crédito,Sí,4)*. Si el parámetro *MinSupport* está por encima del 20% y no supera el 40%, el ítem *(Crédito,No,1)* se elimina de la tabla, con lo que se obtiene el conjunto *L[1]* con cinco ítems relevantes.

El conjunto real completo de instrucciones SQL que se ejecutarían es el siguiente:

①
```
INSERT INTO L1
SELECT 1, Datos.Aval, COUNT(*)
FROM Datos
GROUP BY Datos.Aval
```

②
```
INSERT INTO L1
SELECT 2, Datos.Nómina, COUNT(*)
FROM Datos
GROUP BY Datos.Nómina
```

③
```
INSERT INTO L1
SELECT 3, Datos.Crédito, COUNT(*)
FROM Datos
GROUP BY Datos.Crédito
```

④
```
DELETE FROM L1
WHERE support < MinSupport
```

Las columnas de la tabla de ejemplo se han codificado de la siguiente manera: el 1 corresponde a *"Aval"*, el 2 referencia a *"Nómina"* y el 3 concierne a la columna *"Crédito"*.

La primera instrucción introduce en la tabla *L[1]* los ítems *Aval:Sí* y *Aval:No* con sus relevancias (2 y 3 respectivamente). La segunda añade las tuplas *(Nómina,No,2)* y *(Nómina,Sí,3)* a la tabla de itemsets relevantes L[1] y la tercera completa esta tabla con las tuplas *(Crédito,No,1)* y *(Crédito,Sí,4)*.

Finalmente, si el número de tuplas fijado por MinSupport es igual a 2, al ejecutar la sentencia ④ se eliminará de la tabla la tupla *(Crédito,No,1)*. Por lo tanto, la tabla de ítems relevantes *L[1]* es:

ATTR1	VAL1	SUPPORT
1	No	3
1	Sí	2
2	No	2
2	Sí	3
3	Sí	4

Tabla L[1]

GENERACIÓN DE CANDIDATOS: C[K]

En el algoritmo *Apriori*, la generación del conjunto de candidatos *C[k]* se realiza directamente a partir del conjunto de itemsets relevantes *L[k-1]*. En primer lugar se generan posibles candidatos a partir del producto cartesiano *L[k-1]*L[k-1]* imponiendo la restricción de que los k-2 primeros items de los elementos de *L[k-1]* han de coincidir. A continuación se eliminan del conjunto de candidatos aquellos itemsets que contienen algún *(k-1)*-itemset que no se encuentre en *L[k-1]*.

En el algoritmo ***t*** se utiliza una técnica similar. Inicialmente se incluyen en el conjunto de candidatos aquellos itemsets formados uniendo parejas de itemsets de *L[k-1]* cuyos *k-2* primeros items coincidan y sus últimos ítems corresponda a distintos atributos de la relación (distintas columnas de la tabla). Una vez hecho esto, se eliminan del conjunto de candidatos aquéllos itemsets que contienen algún *(k-1)*-itemset que no esté en *L[k-1]*.

La diferencia existente entre los procesos de generación de candidatos de los algoritmos *Apriori* y ***t*** proviene del concepto de ítem considerado. En el algoritmo ***t***, un ítem es un par *a:v* mientras que en Apriori un ítem era, simplemente, un valor *v*.

```
// Join

INSERT INTO C[k]
SELECT p.ATTR1, p.VAL1, .., p.ATTR(k-1), p.VAL(k-1), q.ATTR(k-1), q.VAL(k-1)
FROM L[k-1] p, L[k-1] q
WHERE p.ATTR1=q.ATTR1 AND p.VAL1=q.VAL1
  AND ...
  AND p.ATTR(k-2)=q.ATTR(k-2) AND p.VAL(k-2)=q.VAL(k-2)
  AND p.ATTR(k-1)<q.ATTR(k-1)

// Prunning

for (i=1; i<k-2; i++)

    DELETE
    FROM C[k]
    WHERE NOT EXISTS
    ( SELECT *
      FROM L[k-1]
      WHERE C[k].ATTR1=L[k-1].ATTR1 AND C[k].VAL1=L[k-1].VAL1
      AND ...
      AND C[k].ATTR(i-1)=L[k-1].ATTR(i-1) AND C[k].VAL(i-1)=L[k-1].VAL(i-1)
      AND C[k].ATTR(i+1)=L[k-1].ATTR(i+1) AND C[k].VAL(i+1)=L[k-1].VAL(i+1)
      AND ...
      AND C[k].ATTR(k)=L[k-1].ATTR(k) AND C[k].VAL(k)=L[k-1].VAL(k)
    )
```

En el proceso de poda no es necesario comprobar que el itemset contiene los itemsets {attr1:val1..attr(k-1):val(k-1)} y {attr1:val1..attr(k-2):val(k-2) attr(k):val(k)} ya que éstos son los itemsets de los que se obtuvo el k-itemset candidato en el *join*.

Ejemplos del proceso de generación de candidatos:

☞ Generación de *C[2]* a partir de *L[1]*:

```
CREATE TABLE C2 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER);

INSERT INTO C2
SELECT p.attr1,p.val1,q.attr1,q.val1
FROM L1 p, L1 q
WHERE p.attr1<q.attr1;
```

☞ Generación de *C[3]* a partir de *L[2]*:

```
CREATE TABLE C3 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER,
                  attr3 INTEGER, val3 INTEGER);

INSERT INTO C3
SELECT p.attr1,p.val1,p.attr2,p.val2,q.attr2,q.val2
FROM L2 p, L2 q
WHERE p.attr1=q.attr1 AND p.val1=q.val1
  AND p.attr2<q.attr2;

DELETE FROM C3
WHERE NOT EXISTS ( SELECT * FROM L2
                   WHERE C3.attr2=L2.attr1 AND C3.val2=L2.val1
                     AND C3.attr3=L2.attr2 AND C3.val3=L2.val2);
```

☞ Generación de *C[4]* a partir de *L[3]*:

```
CREATE TABLE C4 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER,
                  attr3 INTEGER, val3 INTEGER,
                  attr4 INTEGER, val4 INTEGER);

INSERT INTO C4
SELECT p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3,q.attr3,q.val3
FROM L3 p, L3 q
WHERE p.attr1=q.attr1 AND p.val1=q.val1
  AND p.attr2=q.attr2 AND p.val2=q.val2
  AND p.attr3<q.attr3;

DELETE FROM C4
WHERE NOT EXISTS ( SELECT * FROM L3
                   WHERE C4.attr2=L3.attr1 AND C4.val2=L3.val1
                     AND C4.attr3=L3.attr2 AND C4.val3=L3.val2
                     AND C4.attr4=L3.attr3 AND C4.val4=L3.val3);
DELETE FROM C4
WHERE NOT EXISTS ( SELECT * FROM L3
                   WHERE C4.attr1=L3.attr1 AND C4.val1=L3.val1
                     AND C4.attr3=L3.attr2 AND C4.val3=L3.val2
                     AND C4.attr4=L3.attr3 AND C4.val4=L3.val3);
```

OBTENCIÓN DE LOS CONJUNTOS DE ITEMSETS RELEVANTES: *L[K] K≥2*

A partir del conjunto de candidatos *C[k]* se puede obtener fácilmente el conjunto de itemsets relevantes *L[k]*. Nos basta con realizar una pasada por la base de datos para obtener la relevancia de cada candidato y no incluir en el conjunto *L[k]* aquellos candidatos cuya relevancia no alcance el umbral preestablecido *MinSupport*.

Expresar en SQL la sentencia necesaria para obtener la relevancia de los itemsets candidatos es algo complejo, debido a la estructura de datos empleada para almacenar los conjuntos de itemsets:

```
// Relevancia de los itemsets candidatos

INSERT INTO L[k]
SELECT p.ATTR1, p.VAL1 .. p.ATTR(k-1), p.VAL(k-1), COUNT(*)
FROM C[k] p, Datos q
WHERE (    (p.ATTR1=1 AND p.VAL1=q.columna(1))
        OR ...
        OR (p.ATTR1=n AND p.VAL1=q.columna(n)) )
  AND ...
  AND (    (p.ATTRk=1 AND p.VALk=q.columna(1))
        OR ...
        OR (p.ATTRk=n AND p.VALk=q.columna(n)) )
GROUP BY p.ATTR1, p.VAL1 .. p.ATTR(k-1), p.VAL(k-1)


// Eliminación de los itemsets no relevantes

DELETE FROM L[k]
WHERE support < MinSupport
```

Este proceso produce el mismo resultado que aplicar lo siguiente a cada uno de los itemsets *{attr1:val1 ... attrK:valK}* del conjunto *C[k]*:

```
SELECT COUNT(*)
FROM Datos
WHERE columna(attr1)=val1 AND ... AND columna(attrK)=valK

Si COUNT≥MinSupport

      INSERT INTO L[k]
      VALUES (attr1,val1,..,attrK,valK,COUNT)
```

Ejemplos del proceso de obtención de los itemsets relevantes:

Se utiliza la misma tabla que en ejemplos anteriores. La tabla corresponde a una muestra de la base de datos de una entidad de crédito ficticia, tiene tres columnas (aval, nómina y crédito) y cinco tuplas.

☞ Proceso de obtención de *L[2]* a partir de *C[2]*:

```
CREATE TABLE L2 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER, support INTEGER );

INSERT INTO L2
SELECT p.attr1,p.val1,p.attr2,p.val2,COUNT(*)
FROM C2 p, DATOS q
WHERE (  (p.ATTR1=1 AND p.VAL1=q.AVAL)
      OR (p.ATTR1=2 AND p.VAL1=q.NÓMINA)
      OR (p.ATTR1=3 AND p.VAL1=q.CRÉDITO))
  AND (  (p.ATTR2=1 AND p.VAL2=q.AVAL)
      OR (p.ATTR2=2 AND p.VAL2=q.NÓMINA)
      OR (p.ATTR2=3 AND p.VAL2=q.CRÉDITO))
GROUP BY p.attr1,p.val1,p.attr2,p.val2;

DELETE FROM L2 WHERE support < 3;
```

☞ Proceso de obtención de *L[3]* a partir de *C[3]*:

```
CREATE TABLE L3 ( attr1 INTEGER, val1 INTEGER,
                  attr2 INTEGER, val2 INTEGER,
                  attr3 INTEGER, val3 INTEGER, support INTEGER );

INSERT INTO L3
SELECT p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3,COUNT(*)
FROM C3 p, DATOS q
WHERE (  (p.ATTR1=1 AND p.VAL1=q.AVAL)
      OR (p.ATTR1=2 AND p.VAL1=q.NÓMINA)
      OR (p.ATTR1=3 AND p.VAL1=q.CRÉDITO))
  AND (  (p.ATTR2=1 AND p.VAL2=q.AVAL)
      OR (p.ATTR2=2 AND p.VAL2=q.NÓMINA)
      OR (p.ATTR2=3 AND p.VAL2=q.CRÉDITO))
  AND (  (p.ATTR3=1 AND p.VAL3=q.AVAL)
      OR (p.ATTR3=2 AND p.VAL3=q.NÓMINA)
      OR (p.ATTR3=3 AND p.VAL3=q.CRÉDITO))
GROUP BY p.attr1,p.val1,p.attr2,p.val2,p.attr3,p.val3;

DELETE FROM L3 WHERE support < 3;
```

Ya que la tabla sólo tiene tres columnas diferentes, la generación de conjuntos de itemsets candidatos con $k \geq 4$ carece de sentido: los itemsets tendrán, como mucho, tres ítems.

Aunque en los ejemplos las tablas se han denominado igual que los conjuntos que representan (con la salvedad de que los corchetes no son caracteres permitidos para los identificadores de las tablas), esto no resulta adecuado en una implementación real del algoritmo.

En primer lugar, es posible que un mismo usuario quiera mantener los conjuntos de itemsets relevantes obtenidos (p.ej. para analizar los datos a posteriori o para fusionar de algún modo resultados obtenidos independientemente). Si todos los k-itemsets relevantes provenientes de distintas aplicaciones del algoritmo (o incluso de aplicaciones sucesivas del algoritmo sobre la misma tabla cambiando el parámetro *MinSupport*) se introducen en la misma tabla, el caos es inmediato.

Por otra parte, si el algoritmo debe funcionar en un servidor de bases de datos relacionales, sería deseable que permitiese acceso concurrente a la base de datos desde distintas instancias de la aplicación de *Data Mining*.

Por lo tanto, es aconsejable (por no decir necesario) idear un mecanismo que permita la ejecución concurrente del algoritmo ***t*** sobre distintas tablas o sobre la misma tabla usando parámetros diferentes.

Un mecanismo sencillo y eficaz para solucionar este problema es denominar *RAKnnLk* y *RAKnnCk* a las tablas que representan los conjuntos *L[k]* y *C[k]* respectivamente. El valor *nn* será único para cada ejecución del algoritmo y, de esta forma, se permite la ejecución concurrente del mismo desde varias aplicaciones conectadas al mismo servidor.

Para asignar dinámicamente el valor *nn* que identifica unívocamente las tablas correspondientes a una aplicación (al igual que el PID identifica a un proceso) se utiliza una tabla denominada *RAK* [*Remote Access Key*] en la que se registran el usuario, la tabla que manipula y el código del permiso concedido (el valor *nn* de los identificadores de las tablas).

La tabla usada anteriormente podría estar codificada de la siguiente forma, donde los valores “Sí” y “No” han sido sustituidos por 1 y 0:

Aval	Nómina	Crédito
0	0	0
0	0	1
0	1	1
1	1	1
1	1	1

Datos

La codificación realizada resulta conveniente ya que es mucho más eficiente manejar directamente enteros que tener que manipular cadenas (p.ej. la comparación de cadenas de caracteres es mucho más costosa que la comparación de enteros, más aún si tenemos que considerar características del lenguaje como signos de acentuación, diéresis, etc.).

El algoritmo *t* genera en primer lugar la tabla L1, donde las columnas de la tabla están identificadas por su posición (1 es *“Aval”*, 2 es *“Nómina”* y 3 es *“Crédito”*).

ATTR1	VAL1	SUPPORT
1	0	3
1	1	2
2	0	2
2	1	3
3	1	4

Tabla *L1*

Aclaración: Al generar L1 el item *3:0* [*Crédito:No*] ha sido eliminado de la tabla al establecerse *MinSupport* en 2 y ser 1 la relevancia del ítem: *support(3:0) = 1 < MinSupport.*

A partir de la tabla *L1* se genera la tabla de candidatos *C2*:

ATTR1	VAL1	ATTR2	VAL2
1	0	2	0
1	0	2	1
1	0	3	1
1	1	2	0
1	1	2	1
1	1	3	1
2	0	3	1
2	1	3	1

De *C2* se obtiene con facilidad *L2*:

ATTR1	VAL1	ATTR2	VAL2	SUPPORT
1	0	2	0	2
1	0	3	1	2
1	1	2	1	2
1	1	3	1	2
2	1	3	1	3

Repitiendo el proceso se obtiene *C3* a partir de *L2*:

ATTR1	VAL1	ATTR2	VAL2	ATTR3	VAL3
1	0	2	0	3	1
1	1	2	1	3	1

Finalmente, a partir de *C3* obtenemos *L3*:

ATTR1	VAL1	ATTR2	VAL2	ATTR3	VAL3	SUPPORT
1	1	2	1	3	1	2

El proceso de obtención de itemsets ha finalizado por dos motivos: en la tabla sólo hay 3 columnas (no puede haber itemsets en L4) y en L3 sólo queda un elemento. Harían falta al menos 2 para que existiese la posibilidad de que hubiese candidatos en C4, cuatro para que hubiese algún 4-itemset relevante y, además, la tabla no tiene tantas columnas.

2.2 Reglas de asociación

Como ya se comentó en la introducción, una regla de asociación es una implicación de la forma X⇒Y donde X e Y son itemsets de intersección vacía. La fiabilidad [*confidence*] de la regla de asociación X⇒Y es la proporción de tuplas que incluyen X y contienen también a Y: *support(X∪Y)/support(X)*. La relevancia [*support*] de la regla de asociación X⇒Y es la proporción de tuplas de la base de datos que contienen tanto a X como a Y: *support(X∪Y)*.

Las reglas de asociación se obtienen fácilmente a partir de los conjuntos de itemsets relevantes. Para cada itemset relevante *i* obtenemos todos sus subconjuntos propios *s*. De cada subconjunto *s* (que ya sabemos que es relevante al ser *i* relevante) se obtiene una regla de la forma *s⇒(i-s)*. La fiabilidad de esta regla es igual al cociente *support(i)/support(s)*. Si la fiabilidad de la regla no alcanza el umbral prefijado *MinConfidence* entonces la descartamos; si lo alcanza, la conservamos.

Un método directo y sencillo de resolver el problema quedaría descrito en lenguaje algorítmico de la siguiente forma:

```
Para cada itemset relevante lk = {attr1:val1 .. attrK:valK} ∈ L[k], k≥2

     // Generar reglas a partir de lk

     Para i de 1 a k-1

          // Búsqueda en L[i] de subconjuntos propios de lk

          SELECT * FROM L[i]
          WHERE   (  (ATTR1=attr1 AND VAL1=val1)
                  OR ...
                  OR (ATTR1=attrK AND VAL1=valK) )
             AND  ...
             AND  (  (ATTRi=attr1 AND VALi=val1)
                  OR ...
                  OR (ATTRi=attrK AND VALi=valK) )

          Para cada ítem li que verifique la consulta SQL anterior

               Si support(lk)/support(li) ≥ MinConfidence

                  Regla li ⇒ (lk-li)  [support(lk)/support(li)]
```

Obviamente el algoritmo descrito no es muy eficiente. Para cada k-itemset relevante se realizan (k-1) consultas a la base de datos, lo que puede consumir un tiempo considerable. Esto se deriva de la representación de los itemsets (están almacenados en tablas).

Para mejorar la eficiencia del algoritmo expuesto, se pueden generar los subconjuntos de un itemset en un orden diferente. Si un subconjunto *a* de un itemset *l* genera una regla $a \Rightarrow (l-a)$ que no tiene la fiabilidad mínima requerida, entonces ningún subconjunto propio *s* del itemset *a* generará reglas que alcancen la fiabilidad mínima *MinConfidence*, ya que $support(s) > support(a)$ y, por lo tanto, $support(l)/support(s) \leq support(l)/support(a)$. La fiabilidad de la regla $a \Rightarrow (l-a)$, que no llegaba a *MinConfidence*, será igual o superior a la de la regla $s \Rightarrow (l-s)$ derivada de *s*.

Si no se generan reglas derivadas de un k-itemset relevante l_k y un subconjunto propio de éste con n items, tampoco se obtendrán reglas derivadas de l_k y elementos de $L[n-1]$:

```
Para cada itemset relevante l_k = {attr1:val1 .. attrK:valK} ∈ L[k], k≥2

     // Generar reglas a partir de l_k

     i = k

     Repetir

            i = i-1

            SELECT *
            FROM L[i]
            WHERE (  (ATTR1=attr1 AND VAL1=val1)
                  OR ...
                  OR (ATTR1=attrK AND VAL1=valK) )
              AND ...
              AND (  (ATTRi=attr1 AND VALi=val1)
                  OR ...
                  OR (ATTRi=attrK AND VALi=valK) )

            Para cada ítem l_i que verifique la consulta SQL anterior

                 Si support(l_k)/support(l_i) ≥ MinConfidence

                    Regla l_i ⇒ (l_k-l_i)  [support(l_k)/support(l_i)]

     Mientras i>1
            Y se hayan obtenido reglas con la fiabilidad mínima
```

Esta variante del algoritmo inicial permite que en ocasiones se realicen menos de *k-1* consultas para cada k-itemset relevante, lo que supone una pequeña mejora en la eficiencia del método.

En la fase de la generación de reglas de asociación se puede apreciar con claridad cómo almacenar los itemsets en tablas no resulta muy adecuado para elaborar algoritmos eficientes.

EJEMPLO

Utilicemos una vez más la base de datos de la entidad de crédito para ilustrar el funcionamiento del algoritmo. Como vimos en el apartado anterior, los conjuntos de itemsets relevantes que se obtenían a partir de la tabla de ejemplo (con *MinSupport* 40%) son:

ATTR1	VAL1	SUPPORT
1	0	3
1	1	2
2	0	2
2	1	3
3	1	4

Tabla *L1*

ATTR1	VAL1	ATTR2	VAL2	SUPPORT
1	0	2	0	2
1	0	3	1	2
1	1	2	1	2
1	1	3	1	2
2	1	3	1	3

Tabla L2

ATTR1	VAL1	ATTR2	VAL2	ATTR3	VAL3	SUPPORT
1	1	2	1	3	1	2

Tabla L3

Nota: Los atributos 1, 2 y 3 corresponden a las columnas “Aval”, “Nómina” y “Crédito” de la tabla original. Los valores “Sí” y “No” han sido reemplazados por 1 y 0.

El algoritmo para la generación de reglas de asociación requiere recorrer todos los itemsets relevantes que contengan al menos dos items. En este caso, se ha de comprobar la generación de reglas de asociación para los 5 miembros de *L[2]* y el único elemento de *L[3]*.

En este ejemplo se supondrá que *MinConfidence* es igual a *0.5* (el 50%):

Recorrido de L[2]:

✯ Itemset {1:0, 2:0} ≡ {Aval:No, Nómina:No}

Se obtienen los subconjuntos propios del itemset incluidos en L[1]: {1:0} y {2:0}
De ellos se derivan las dos reglas:

☞ {1:0} ⇒ {2:0} [2/3]
☞ {2:0} ⇒ {1:0} [2/2]

✯ Itemset {1:0, 3:1} ≡ {Aval:No, Crédito:Sí}

Siguiendo el mismo proceso se obtienen las reglas:

☞ {1:0} ⇒ {3:1} [2/3]
☞ {3:1} ⇒ {1:0} [2/4]

✯ Itemset {1:1, 2:1} ≡ {Aval:Sí, Nómina:Sí}

Esta vez se generan:

☞ {1:1} ⇒ {2:1} [2/2]
☞ {2:1} ⇒ {1:1} [2/3]

✯ Itemset {1:1, 3:1} ≡ {Aval:Sí, Crédito:Sí}

Se derivan las siguientes reglas:

☞ {1:1} ⇒ {3:1} [2/2]
☞ {3:1} ⇒ {1:1} [2/4]

✯ Itemset {2:1, 3:1} ≡ {Nómina:Sí, Crédito:Sí}

Se comprueba que se verifican las reglas:

☞ {2:1} ⇒ {3:1} [3/3]
☞ {3:1} ⇒ {2:1} [3/4]

Recorrido de L[3]:

✯ Itemset {1:1, 2:1, 3:1} ≡ {Aval:Sí, Nómina:Sí, Crédito:Sí}

Se comienza por los subconjuntos del itemset pertenecientes a *L[2]*:

☞ {1:1, 2:1} ⇒ {3:1} [2/2]
☞ {1:1, 3:1} ⇒ {2:1} [2/2]
☞ {2:1, 3:1} ⇒ {1:1} [2/3]

Como hemos generado reglas derivadas de elementos L[2] no podemos conocer a priori si se generarán o no reglas derivadas de items de L[1], por lo que no nos queda más remedio que obtener los subconjuntos de {1:1, 2:1, 3:1} incluidos en L[1] y comprobar si las reglas generadas alcanzan la fiabilidad mínima requerida por el usuario:

☞ {1:1} ⇒ {2:1, 3:1} [2/2]
☞ {2:1} ⇒ {1:1, 3:1} [2/3]
☞ {3:1} ⇒ {1:1, 2:1} [2/4]

El recorrido por los itemsets relevantes ha finalizado y hemos obtenido 16 reglas de asociación derivadas de la tabla de ejemplo. Las reglas de asociación extraídas, enumeradas de una forma más legible, son las siguientes:

Nº	Regla de asociación	Fiabilidad
1	Aval:No ⇒ Nómina:No	2/3 66%
2	Nómina:No ⇒ Aval:No	2/2 100%
3	Aval:No ⇒ Crédito:Sí	2/3 66%
4	Crédito:Sí ⇒ Aval:No	2/4 50%
5	Aval:Sí ⇒ Nómina:Sí	2/2 100%
6	Nómina:Sí ⇒ Aval:Sí	2/3 66%
7	Aval:Sí ⇒ Crédito:Sí	2/2 100%
8	Crédito:Sí ⇒ Aval:Sí	2/4 50%
9	Nómina:Sí ⇒ Crédito:Sí	3/3 100%
10	Crédito:Sí ⇒ Nómina:Sí	3/4 75%
11	Aval:Sí Nómina:Sí ⇒ Crédito:Sí	2/2 100%
12	Aval:Sí Crédito:Sí ⇒ Nómina:Sí	2/2 100%
13	Nómina:Sí Crédito:Sí ⇒ Aval:Sí	2/3 66%
14	Aval:Sí ⇒ Nómina:Sí Crédito:Sí	2/2 100%
15	Nómina:Sí ⇒ Aval:Sí Crédito:Sí	2/3 66%
16	Crédito:Sí ⇒ Aval:Sí Nómina:Sí	2/4 50%

3. Algoritmo T

Generalmente no nos encontraremos la base de datos de la que queremos extraer conocimiento en un formato idóneo. No podremos aplicar directamente algoritmos de *Data Mining* sin antes tener que realizar alguna etapa de preprocesamiento de la información almacenada.

Cuando estemos manejando grandes bases de datos, quizá no dispongamos de los recursos suficientes para tener una copia de la base de datos completa que podamos modificar libremente. Aunque dispusiésemos de los recursos necesarios, puede que nos interese trabajar directamente sobre la base de datos original para obtener resultados precisos y evitar errores que se producirían al manejar una copia de la base de datos que puede que no se corresponda con el estado actual de la base de datos real.

Un mecanismo sencillo que nos permite obtener cierta flexibilidad a la hora de manipular una base de datos consiste en disponer de información adicional para agrupar valores cuyo significado sea equivalente o similar. Podríamos agrupar valores numéricos en intervalos sin tener que alterar la base de datos real o establecer equivalencias entre valores de otros tipos (p.ej. el apellido Fernández podría aparecer de múltiples formas cuando en realidad siempre nos estamos refiriendo a lo mismo: *Fernández, FERNÁNDEZ, Fernandez, FERNANDEZ, Fdez., FDEZ...*).

El algoritmo *T* no es más que una versión ampliada del algoritmo *t* en la cual se permite agrupar los valores de un atributo estableciendo cualquier tipo de taxonomía.

Para agrupar los valores en dominios (que no han de ser disjuntos) el algoritmo *T* utiliza tablas auxiliares: una general para identificar a qué atributo se refiere cada tabla extra y una tabla adicional por atributo que contiene información acerca de los valores que toma cada atributo.

Tabla ATTR (RAKnnATTR en la práctica[*]):

```
CREATE TABLE ATTR
             ( id   INTEGER PRIMARY KEY,
               attr CHAR(32),
               tipo INTEGER )
```

Esta tabla contiene una entrada por cada una de las columnas de la tabla de datos utilizadas en el proceso de extracción de reglas de asociación. Si no nos interesa alguna columna (por no ser relevante para la obtención de las reglas), simplemente no la incluiremos en esta tabla.

Tablas ATTRi (RAKnnATTRi en la práctica):*

Estas tablas especifican los dominios en los que se distribuyen los valores de las distintas columnas de las tablas. Las tablas *ATTRi* tienen distinto formato para atributos numéricos y no numéricos por lo que se hace necesaria la tabla *ATTR* ya descrita (para especificar el tipo del atributo en cuestión, si es o no numérico):

ATRIBUTOS NO NUMÉRICOS

```
CREATE TABLE ATTRi
             ( id  INTEGER,
               val CHAR(columna[i].longitud) )
```

ATRIBUTOS NUMÉRICOS

```
CREATE TABLE ATTRi
             ( id  INTEGER,
               inf NUMBER(columna[i].longitud,columna[i].decimales),
               sup NUMBER(columna[i].longitud,columna[i].decimales))
```

Los dominios atómicos para los atributos numéricos directamente se representan mediante intervalos *[inf, sup]*. Como es lógico, para representar un dominio con un único valor basta con especificar *[valor, valor]* como intervalo.

* DETALLES DE IMPLEMENTACIÓN: Al igual que en la implementación del algoritmo t, los identificadores de las tablas han de ser tales que se permita la existencia y manipulación concurrente de conjuntos de itemsets provenientes de distintas ejecuciones del algoritmo *T*. El valor *nn* será único para cada ejecución del algoritmo y, para asignar dinámicamente este valor, se utiliza una tabla denominada *RAK* [*Remote Access Key*] en la que se registran el usuario, la tabla que manipula y el código del permiso concedido (el valor *nn*).

Con el esquema utilizado en el algoritmo *T* se permite cualquier taxonomía: un valor asociado a un ID único y específico para ese valor (un dominio con un solo elemento), varios valores de un atributo con el mismo ID (valores agrupados en dominios) o el mismo valor en distintas tuplas de las tablas *ATTRi* con distintos IDs (valores pertenecientes simultáneamente a distintos dominios, vg: jerarquía de dominios).

El algoritmo *T* se complica respecto al algoritmo *t* ya que debemos tener muy en cuenta el contenido de las tablas *ATTRi*, que ha de enfrentarse a los valores existentes en la tabla de datos.

A pesar de que las etapas de generación de candidatos (una de las etapas que más tiempo consume) y extracción de las reglas de asociación del algoritmo *T* son *idénticas* a las utilizadas en el algoritmo *t*, la obtención de los conjuntos de itemsets relevantes es más compleja computacionalmente en *T* que en el algoritmo *t*.

Donde antes bastaba con comprobar la condición ***columna[i]=valor*** ahora tenemos que encontrar los valores de ID tales que ***ATTRi.val=columna[i] AND ATTRi.id=ID*** (que se convierte en ***ATTRi.inf≤columna[i] AND ATTRi.sup≥columna[i] AND ATTRi.id=ID*** en el caso de atributos numéricos).

Como muestra de la complejidad añadida, el algoritmo *t* ejecutaba la siguiente orden SQL en la obtención de *L[2]* a partir de *C[2]* para la tabla usada en anteriores ejemplos:

```
INSERT INTO L2
SELECT p.attr1,p.val1,p.attr2,p.val2,COUNT(*)
FROM C2 p, DATOS q
WHERE (  (p.ATTR1=1 AND p.VAL1=q.AVAL)
      OR (p.ATTR1=2 AND p.VAL1=q.NÓMINA)
      OR (p.ATTR1=3 AND p.VAL1=q.CRÉDITO))
  AND (  (p.ATTR2=1 AND p.VAL2=q.AVAL)
      OR (p.ATTR2=2 AND p.VAL2=q.NÓMINA)
      OR (p.ATTR2=3 AND p.VAL2=q.CRÉDITO))
GROUP BY p.attr1,p.val1,p.attr2,p.val2;
```

Al usar el algoritmo *T*, la inserción anterior quedaría como sigue:

```
INSERT INTO L2
SELECT p.attr1,p.val1,p.attr2,p.val2,COUNT(*)
FROM C2 p, DATOS q, ATTR1, ATTR2, ATTR3
WHERE (  (p.ATTR1=1 AND p.VAL1=ATTR1.id)
      OR (p.ATTR1=2 AND p.VAL1=ATTR2.id)
      OR (p.ATTR1=3 AND p.VAL1=ATTR3.id))
  AND (  (p.ATTR2=1 AND p.VAL2=ATTR1.id)
      OR (p.ATTR2=2 AND p.VAL2=ATTR2.id)
      OR (p.ATTR2=3 AND p.VAL2=ATTR3.id))
  AND q.AVAL=ATTR1.val                                  // ATTR1 no numérico
  AND q.NÓMINA>=ATTR2.inf AND q.NÓMINA<=ATTR2.sup       // ATTR2 numérico
  AND q.CRÉDITO>=ATTR3.inf AND q.CRÉDITO<=ATTR3.sup     // ATTR3 numérico
GROUP BY p.attr1,p.val1,p.attr2,p.val2;
```

La complejidad extra introducida por el uso de las tablas *ATTRi* para la especificación de los dominios de los distintos atributos pone aún más de manifiesto la poca idoneidad del uso de tablas para la representación de conjuntos de itemsets. Las ya complejas consultas a la base de datos que habían de realizarse en el algoritmo t se ven ahora dificultadas aún más por la introducción de tablas adicionales (en número proporcional al número de columnas de la tabla). La eficiencia del algoritmo se ve penalizada por el uso de consultas que involucran múltiples tablas y complejas condiciones booleanas.

EJEMPLO

Partamos una vez más de la tabla usada en anteriores ejemplos (la de la entidad de crédito). La información original contenida en la tabla podría ser la siguiente, donde se especifica la nómina del cliente y la cuantía del crédito concedido:

Aval	Nómina	Crédito
No	0	0
No	10.000	25.000
No	75.000	500.000
Sí	250.000	2.000.000
Sí	1.000.000	20.000.000

Para esta tabla de datos, el algoritmo *T* podría generar las siguientes tablas con información acerca de los dominios de los atributos:

ID	ATTR	TIPO
1	Aval	CHAR
2	Nómina	NUMBER
3	Crédito	NUMBER

Tabla *ATTR*

ID	VAL
0	No
1	Sí

Tabla ATTR1: "Aval"

La tabla ATTR1 que describe los dominios del atributo "Aval" es trivial y no necesita más comentarios. Los otros dos atributos, al ser numéricos, se discretizan en intervalos. Como se muestra en la tabla ATTR2, unos ingresos fijos de cincuenta mil pesetas mensuales se consideran el mínimo necesario para admitir que el cliente tiene una nómina estable suficiente para la concesión de créditos. La construcción de la tabla ATTR3 también es trivial:

ID	INF	SUP
0	0	49.999
1	50.000	100.000.000

Tabla ATTR2: "Nómina"

ID	INF	SUP
0	0	0
1	1	100.000.000

Tabla ATTR3: "Crédito"

Una vez establecidos los dominios de los atributos tal como se muestra en las tablas *ATTRi*, el algoritmo *T* se comporta exactamente igual que lo hacía el algoritmo *t*. En este caso es como si tuviésemos como entrada al algoritmo *t* la tabla siguiente:

Aval	Nómina	Crédito
0	0	0
0	0	1
0	1	1
1	1	1
1	1	1

La ejecución del algoritmo *T* genera las misma tablas que se obtenían con el algoritmo *t*:

ATTR1	VAL1	SUPPORT
1	0	3
1	1	2
2	0	2
2	1	3
3	1	4

Tabla *L1*

ATTR1	VAL1	ATTR2	VAL2
1	0	2	0
1	0	2	1
1	0	3	1
1	1	2	0
1	1	2	1
1	1	3	1
2	0	3	1
2	1	3	1

Tabla *C2*

ATTR1	VAL1	ATTR2	VAL2	SUPPORT
1	0	2	0	2
1	0	3	1	2
1	1	2	1	2
1	1	3	1	2
2	1	3	1	3

Tabla *L2*

ATTR1	VAL1	ATTR2	VAL2	ATTR3	VAL3
1	0	2	0	3	1
1	1	2	1	3	1

Tabla *C3*

ATTR1	VAL1	ATTR2	VAL2	ATTR3	VAL3	SUPPORT
1	1	2	1	3	1	2

Tabla *L3*

El proceso de obtención de itemsets ha finalizado por dos motivos: en la tabla sólo hay 3 columnas (no puede haber itemsets en L4) y en L3 sólo queda un elemento (harían falta al menos 2 para que existiese la posibilidad de que hubiese candidatos en C4).

Una vez que hemos obtenido todos los conjuntos de itemsets relevantes, el proceso de generación de reglas de asociación es directo. Se utiliza exactamente el mismo método expuesto al desarrollar el algoritmo *t*. Sólo cambia la interpretación de las reglas (trivial en este caso), para lo cual debemos tener en cuenta las tablas auxiliares *ATTRi*:

Nº	Regla de asociación	Fiabilidad
1	Aval:No ⇒ Nómina<50000	2/3 66%
2	Nómina<50000 ⇒ Aval:No	2/2 100%
3	Aval:No ⇒ Crédito:Sí	2/3 66%
4	Crédito:Sí ⇒ Aval:No	2/4 50%
5	Aval:Sí ⇒ Nómina≥50000	2/2 100%
6	Nómina≥50000 ⇒ Aval:Sí	2/3 66%
7	Aval:Sí ⇒ Crédito:Sí	2/2 100%
8	Crédito:Sí ⇒ Aval:Sí	2/4 50%
9	Nómina≥50000 ⇒ Crédito:Sí	3/3 100%
10	Crédito:Sí ⇒ Nómina≥50000	3/4 75%
11	Aval:Sí ∧ Nómina≥50000 ⇒ Crédito:Sí	2/2 100%
12	Aval:Sí ∧ Crédito:Sí ⇒ Nómina≥50000	2/2 100%
13	Nómina≥50000 ∧ Crédito:Sí ⇒ Aval:Sí	2/3 66%
14	Aval:Sí ⇒ Nómina≥50000 ∧ Crédito:Sí	2/2 100%
15	Nómina≥50000 ⇒ Aval:Sí ∧ Crédito:Sí	2/3 66%
16	Crédito:Sí ⇒ Aval:Sí ∧ Nómina≥50000	2/4 50%

4. Algoritmo $\overline{T}$ (TBAR: Tree-Based Association Rule generation)

Creo que nunca veré un poema tan agradable como un árbol

Joyce Kilmer, 1913

Como ya se ha comentado, la estructura de datos utilizada en el algoritmo T para representar los conjuntos de itemsets no resulta adecuada. El algoritmo TBAR cambia la representación interna de los conjuntos de itemsets candidatos y relevantes para conseguir una mejora notable tanto en tiempo de ejecución como en recursos utilizados.

El algoritmo TBAR almacena todos los conjuntos de itemsets, tanto candidatos como relevantes, en una única estructura de datos similar a un árbol de enumeración de subconjuntos [*set-enumeration tree*]. Esta estructura de datos se adapta al problema de la extracción de reglas de asociación para intentar minimizar los cálculos necesarios. Las peculiaridades de la estructura de datos empleada se deben al intento de aprovechar toda la información relativa al dominio del problema de la que disponga a priori. El árbol de itemsets usado y todas sus particularidades se verán con detalle más adelante.

Los algoritmos t y T se caracterizaban por dejar que todo el trabajo lo hiciese el servidor de bases de datos relacionales. En ellos, todas las operaciones se especifican como sentencias SQL que deberán ser interpretadas y ejecutadas por el servidor. En ambos se procura reducir al mínimo la comunicación entre el cliente (la aplicación que ejecuta el algoritmo de extracción de reglas) y el servidor (la base de datos) a costa de que todo el cálculo lo realice el servidor.

Para no sobrecargar en exceso al servidor de bases de datos, que posiblemente tenga que seguir realizando otro tipo de tareas asociadas a la vida cotidiana, sería aconsejable (cuando no necesario) distribuir parte del trabajo entre servidor y cliente. Llevando al extremo esta posibilidad, el servidor de bases de datos se convierte en un simple servidor de archivos secuenciales. El cliente recorre secuencialmente las tablas cuantas veces sea necesario y va realizando todos los cálculos o, al menos, la mayor parte de ellos en la máquina en la que se ejecuta (no en el servidor). Este es un enfoque más práctico y realista, ya que el proceso de *Data Mining* se realiza casi por completo en el terminal del usuario sobrecargando lo menos posible el servidor central (que debe continuar con sus tareas habituales).

A continuación se expondrá una visión general del funcionamiento del algoritmo TBAR, se comentará detalladamente la estructura de datos empleada (el árbol de itemsets), se describirá paso a paso la evolución del algoritmo y se propondrán mejoras del algoritmo encaminadas a incrementar su eficiencia (en tiempo de ejecución y/o consumo de memoria) sin que éstas supongan un cambio de filosofía.

4.1 Visión general

La principal novedad introducida por el algoritmo TBAR es el uso de un árbol para representar todos los conjuntos de itemsets de una forma compacta. Utilizaremos el identificador *set* para referirnos a este árbol de itemsets.

Como en casi todos los algoritmos existentes de extracción de reglas de asociación, el algoritmo TBAR descompone el problema en dos etapas (obtención de itemsets relevantes y generación de reglas de asociación a partir de ellos) que se describen brevemente a continuación:

PRIMERA ETAPA: ITEMSETS RELEVANTES [*(COVERING|FREQUENT|LARGE) ITEMSETS*]

En primer lugar se encuentran todos los itemsets con relevancia por encima de un umbral preestablecido (al que denominamos *MinSupport*).

```
① set.Init (MinSupport)
② set.Relevantes(1);
③ for (k=2; k<=Datos.columnas && set.itemsets(k-1)>=k; k++)
④      set.Candidatos(k);
⑤      if (set.itemsets(k)>0)
⑥          set.Relevantes(k);
```

Comentarios:

① Se crea e inicializa la estructura de datos que contendrá todos los itemsets relevantes. Se trata simplemente de crear un árbol y una tabla hash que permita un acceso eficiente a los conjuntos de itemsets. Inicialmente, el árbol contendrá todos los items diferentes obtenidos a partir de la tabla de datos; es decir, mantendrá el conjunto de 1-itemsets candidatos (el conjunto *C[1]*).

② A partir del conjunto de candidatos *C[1]* se obtiene el conjunto de itemsets relevantes *L[1]*: se obtiene la relevancia de cada ítem de *C[1]* y se eliminan aquéllos cuya relevancia no llegue a *MinSupport*.

③ El bucle se repite mientras se puedan seguir obteniendo k-itemsets relevantes. Como ya se ha visto anteriormente, el número de columnas de la tabla (atributos de la relación) es una cota superior del tamaño de los itemsets: en un mismo itemset no pueden aparecer dos items correspondientes a un mismo atributo [consecuencia de la Primera Forma Normal]. Por otro lado, si el conjunto de itemsets relevantes L[k-1] contiene menos de k itemsets, entonces sabemos que no puede haber ningún k-itemset relevante en L[k]: un itemset de *L[k+e]* incluye $\binom{k+e}{k}$ k-itemsets relevantes y, como caso particular, un itemset relevante de *L[k]* debe incluir exactamente k itemsets de *L[k-1]*: $\binom{k}{k-1} = k$

④ A partir del árbol de itemsets (del conjunto L[k-1] incluido en él para ser más precisos) se genera el conjunto de k-itemsets candidatos, con lo cual la estructura de datos incluirá todos los elementos del conjunto *C[k]*, además de todos los itemsets relevantes de menor tamaño (que ya estaban en *set*).

⑤ Igual que en la condición del bucle comprobábamos el número de (k-1)-itemsets incluidos en la estructura de datos (el conjunto *L[k-1]*), ahora se comprueba la existencia de itemsets candidatos en el conjunto *C[k]*. Al llegar a este punto, el número de candidatos de *C[k]* es igual al número de k-itemsets incluidos en el árbol de itemsets. Obviamente, si no existen candidatos en *C[k]* no podrá existir ningún itemset relevante en *L[k]*.

⑥ Una vez que hemos añadido al árbol de itemsets aquéllos k-itemsets que podrían ser relevantes (el conjunto de candidatos *C[k]*) simplemente hemos de obtener la relevancia de estos itemsets y eliminar del árbol todos aquellos que no superen el umbral establecido por *MinSupport*.

SEGUNDA ETAPA: REGLAS DE ASOCIACIÓN [*ASSOCIATION RULES*]

Una vez obtenidos todos los itemsets relevantes, se generan las reglas de asociación derivadas a partir de ellos. Si *ABCD* y *AB* son dos itemsets relevantes, la regla $AB \Rightarrow CD$ tendrá con seguridad una relevancia mínima (ya que *ABCD* es un itemset relevante), su fiabilidad será igual al cociente *support(ABCD)/support(AB)* y nos quedaremos con ella si este cociente alcanza el valor preestablecido *MinConfidence*.

```
set.Reglas (MinConfidence)
```

En esta fase del algoritmo se debe recorrer la estructura de datos generada (que contiene todos los itemsets relevantes) para obtener todas y cada una de las reglas de asociación cuya fiabilidad llegue a *MinConfidence*.

4.2 Estructura de datos empleada: Árbol de itemsets

Para la representación de los itemsets se ha escogido un árbol como estructura de datos. La estructura de datos utilizada, similar en cierta medida a la usada en el algoritmo *Max-Miner* de Roberto J. Bayardo Jr [*"Efficiently Mining Long Patterns from Databases" ACM PODS'98*], permite representar de forma compacta todos los conjuntos de itemsets.

Al representar todos los itemsets en una única estructura de datos se consigue un ahorro considerable de memoria respecto al almacenamiento independiente de los distintos conjuntos de itemsets que se realiza en algoritmos como *Apriori* de Rakesh Agrawal y Ramakrishnan Skirant [*"Fast Algorithms for Mining Association Rules" IBM Research Report RJ9839, June 1994*]. De hecho *TBAR* se puede considerar un derivado de *Apriori*. El ahorro de memoria es considerable en comparación incluso con el algoritmo *Max-Miner* de Bayardo.

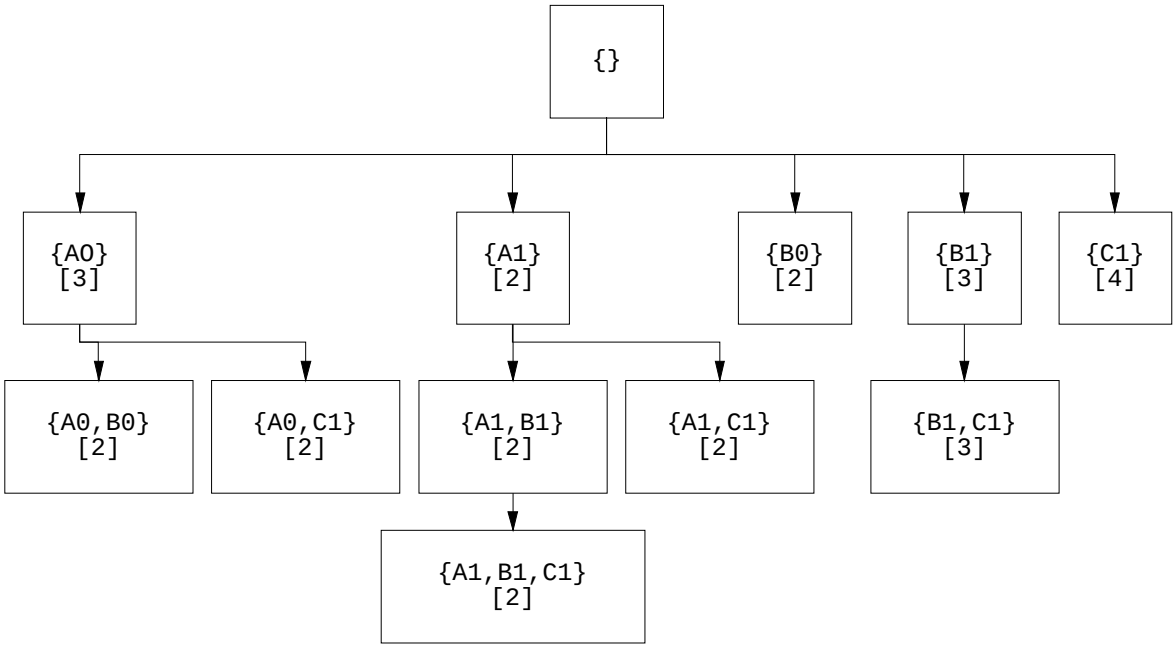
EJEMPLO

Usemos una vez más la tabla de la entidad de crédito y fijemos nuevamente el parámetro *MinSupport* en el 40% (un itemset deberá aparecer al menos en dos de las cinco tuplas para ser considerado relevante). Los distintos valores de los atributos de la tabla se codifican con enteros: el número cero (0) corresponde al valor "no" y el número uno (1) al valor "sí". A continuación se muestra la tabla de ejemplo codificada y los conjuntos de itemsets relevantes que de ella se derivan:

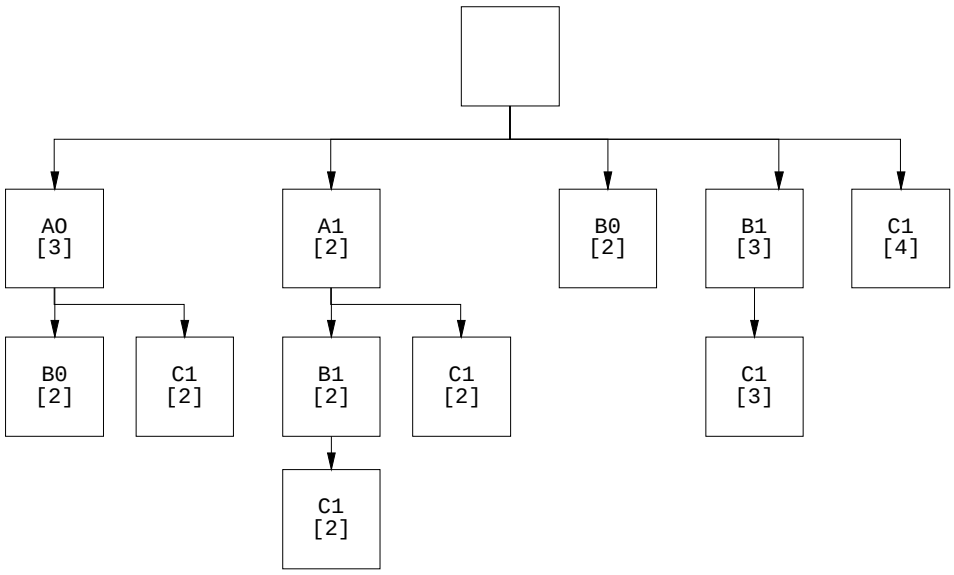
A Aval	B Nómina	C Crédito
0	0	0
0	0	1
0	1	1
1	1	1
1	1	1

Conjunto L[k]	**Itemsets** {items} [support]	**Cardinalidad** #L[k]
L[1]	{A0} [3] {A1} [2] {B0} [2] {B1} [3] {C1} [4]	5
L[2]	{A0,B0} [2] {A0,C1} [2] {A1,B1} [2] {A1,C1} [2] {B1,C1} [3]	5
L[3]	{A1,B1,C1} [2]	1

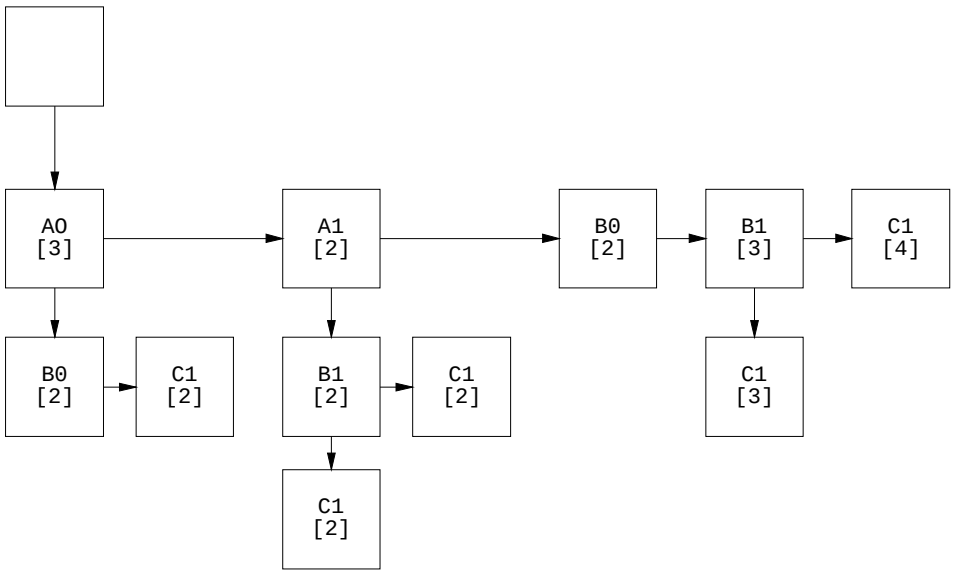
Estos conjuntos de itemsets relevantes se podrían representar en un árbol de enumeración de subconjuntos [*set-enumeration tree*] con el siguiente formato:



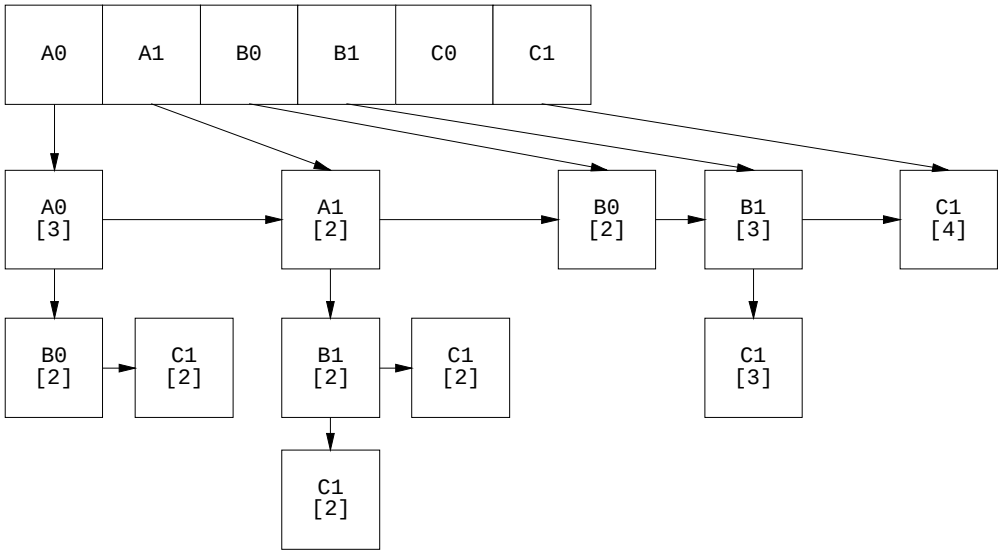
De una forma un poco más compacta podemos representar la misma información con el siguiente árbol de itemsets:



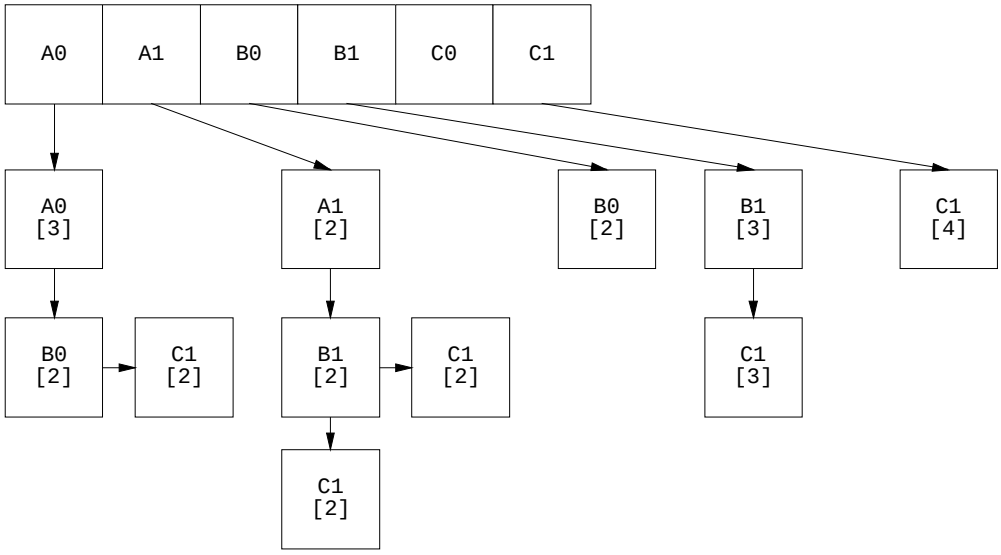
Si utilizamos una representación ***hijo a la izquierda - hermano a la derecha*** el árbol de itemsets se va pareciendo algo más a su implementación final:



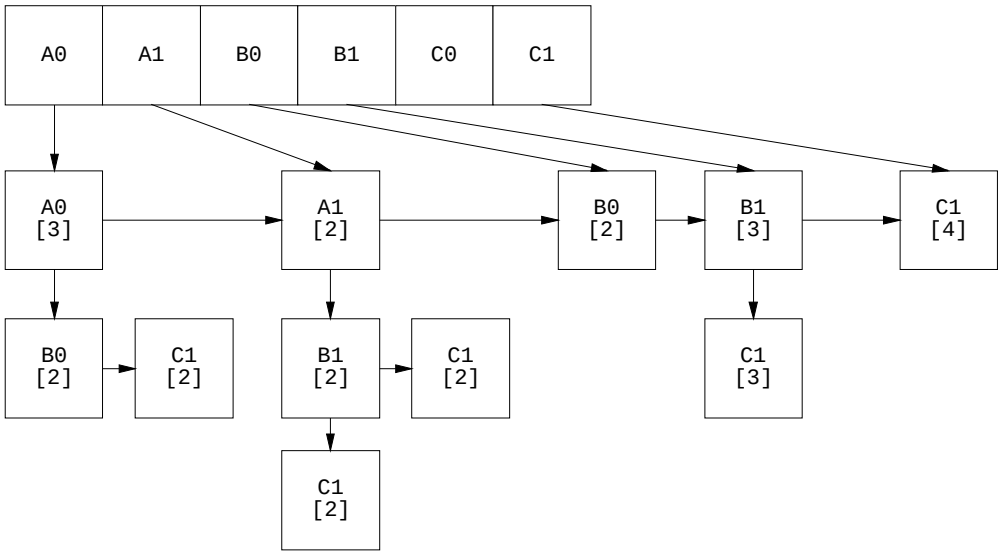
Para optimizar el acceso a los itemsets almacenados en el árbol, podemos introducir una ***tabla hash*** perfecta indexada por parejas ***atributo:dominio*** en la raíz del árbol:



En realidad es como si estuviésemos trabajando con un bosque (un conjunto de árboles). La raíz de cada uno de estos árboles estaría referenciada por una entrada de tabla hash. Cada árbol representaría el conjunto de todos los itemsets (que recordemos se ordenan lexicográficamente) que comienzan por el ítem correspondiente a la entrada de la tabla hash:

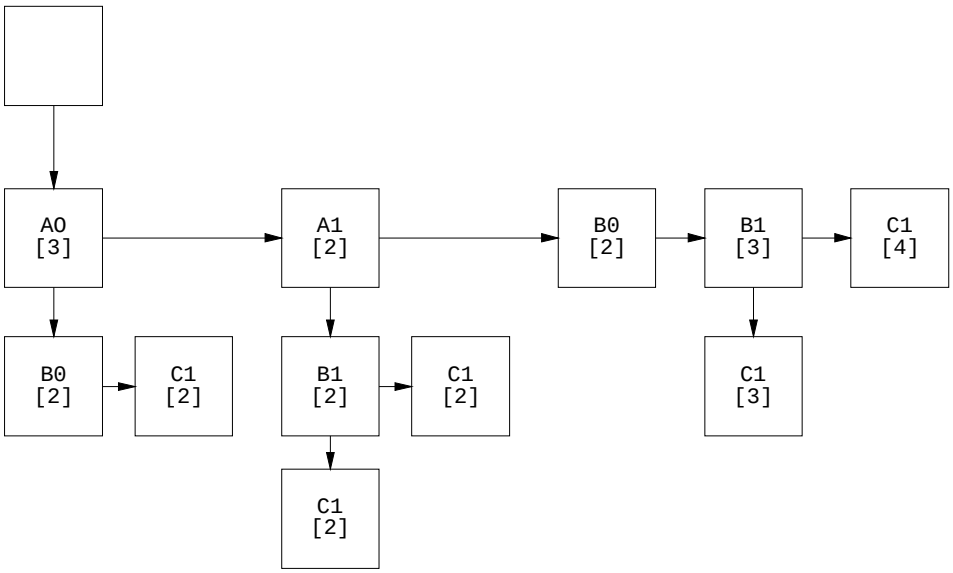


Para facilitar la generación de candidatos *C[2]*, las raíces de estos árboles se pueden enlazar de forma que la estructura de datos se queda como estaba (tabla hash incluida):



Como se puede comprobar fácilmente, la generación de candidatos consiste simplemente en poner como hijos de un nodo sus hermanos a la derecha. Después se obtendrá la relevancia de cada itemset y se eliminarán los nodos correspondientes a itemsets no relevantes (aquéllos cuya relevancia quede por debajo de *MinSupport*).

En el árbol de itemsets, un nodo de profundidad k contiene el k-ésimo item del itemset formado por todos los nodos desde la raíz hasta él (incluido) y la relevancia [*support*] de dicho k-itemset. Cada conjunto *L[k]* viene determinado por el conjunto de nodos del árbol situados a profundidad k. Dado que todos los subconjuntos de un itemset relevante han de ser también relevantes, la estructura de datos resulta bastante adecuada para representar todos los conjuntos de itemsets *L[k]* de una forma lo más compacta posible:



Como se puede apreciar en la figura, para obtener los itemsets candidatos C[k+1] una vez que tenemos los itemsets relevantes hasta L[k] basta con copiar la lista de hermanos a la derecha del k-ésimo nodo de cada k-itemset relevante (descartando automáticamente aquéllos que son del mismo atributo dado que un itemset no puede tener dos items correspondientes a la misma columna de la tabla). Este simple proceso equivale a realizar la operación:

```
INSERT INTO Ck
SELECT p.ATTR1, p.VAL1, .., p.ATTR(k-1), p.VAL(k-1), q.ATTR(k-1), q.VAL(k-1)
FROM L(k-1) p, L(k-1) q
WHERE p.ATTR1=q.ATTR1 AND p.VAL1=q.VAL1
  AND ...
  AND p.ATTR(k-2)=q.ATTR(k-2) AND p.VAL(k-2)=q.VAL(k-2)
  AND p.ATTR(k-1)<q.ATTR(k-1)
```

Una vez contadas las apariciones de los candidatos en la tabla (lo que se consigue con un único recorrido secuencial de ésta), el árbol se poda para quedarnos únicamente con los itemsets relevantes.

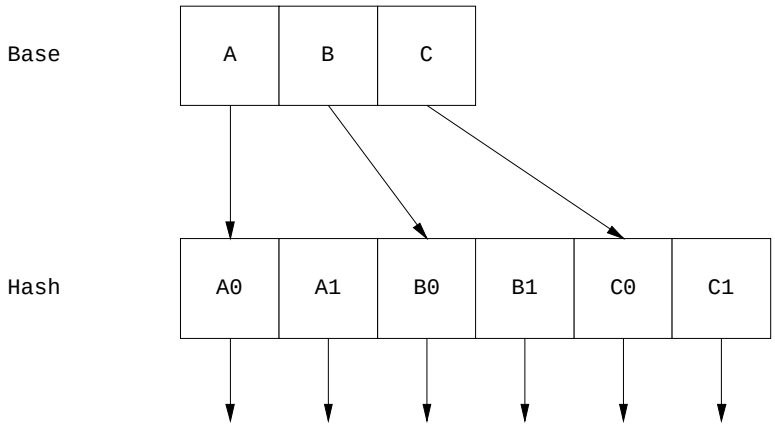
La obtención de las reglas de asociación a partir del árbol de itemsets se complica algo: se deben obtener todos los k-itemsets relevantes ($k \geq 2$) y sus subconjuntos recorriendo de una forma adecuada la estructura de datos, como se verá más adelante.

Árbol de itemsets

El árbol de itemsets, tal como se ha descrito anteriormente (con una tabla hash como raíz) contiene tres variables de instancia: una referencia al primer ítem de *L[1]* (raíz de la estructura en árbol), la tabla hash (un array de items) y un array auxiliar utilizado en la función hash cuyo elemento i-ésimo indica la posición de la tabla hash donde comienzan las posiciones referentes al la columna i-ésima de la tabla. En un lenguaje de alto nivel (como Java) su declaración sería algo así:

Variable de instancia	*Significado*
`NodoItem root`	`Raíz de la estructura en árbol`
`NodoItem hash[]`	`Tabla hash`
`int base[]`	`Usado en la "función hash"`

Para el ejemplo anterior, la estructura la tabla hash perfecta del árbol de itemsets podría interpretarse visualmente como muestra la figura:

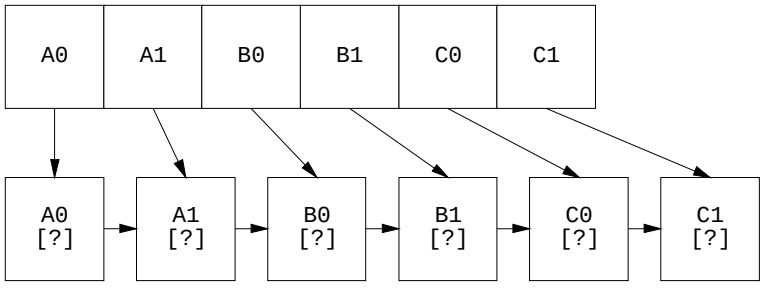


h(atributo:valor) = hash[base[atributo]+valor]

Esta primitiva crea la tabla hash e introduce los items candidatos en el árbol (el conjunto *C[1]*). El tamaño de la tabla hash está determinado por el número de columnas de la tabla (aquéllas que consideramos relevantes en la obtención de reglas de asociación para ser más precisos) y el número de dominios definidos para cada una de esas columnas.

```
// Preparación de la tabla hash

dim   = 0;

for (i=0; i<Datos.columnas; i++)
    base[i] = dim;
    dim += t.columna(i+1).dominios;

// Inicialización: Candidatos C1 a partir de los dominios de la tabla t

for (i=0; i<Datos.columnas; i++)
    for (j=0; j<Datos.columna(i+1).dominios; j++)
        hash[base[i]+j] = NodoItem(i,j);

root = hash[0];
```

Para el ejemplo utilizado en la exposición, tras ejecutar la primitiva *Init* el árbol de itemsets quedaría como muestra el diagrama:

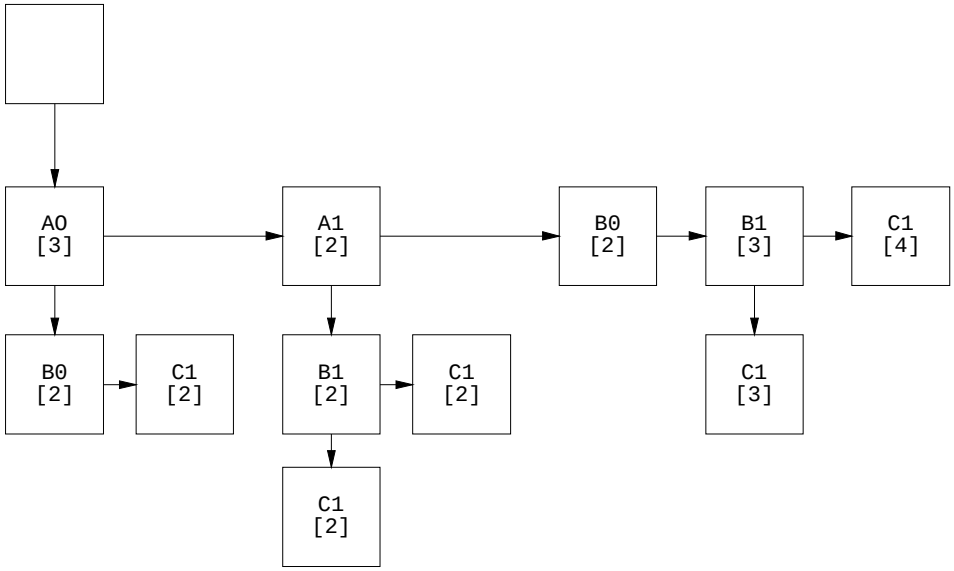


PRIMITIVAS DEL ÁRBOL DE ITEMSETS: ITEMSETS (K)

Esta primitiva se limita a devolver el número de k-itemsets que contiene la estructura de datos en árbol. Esto equivale a contar el número de nodos del árbol situados en el nivel *k* del árbol (nodos a profundidad *k*). Para ello se llama al método *Items* del nodo raíz del árbol (que devuelve el número de descendientes del nodo situados a la profundidad dada):

```
return root.Items(k)
```

vg:



Llamada al método	*Resultado obtenido*
Itemsets(1)	5
Itemsets(2)	5
Itemsets(3)	1

Esta primitiva del TDA se encarga de la obtención de los k-itemsets relevantes (es decir, el conjunto *L[k]*) a partir del conjunto de candidatos *C[k]*.

El proceso de obtención del conjunto de itemsets relevantes *L[k]* una vez conocido el conjunto de itemsets candidatos *C[k]* es trivial: $L[k] = \{ c \mid c \in C[k] \wedge \#c \geq MinSupport \}$

```
① Para cada tupla t de la tabla Datos
②      Count (t,k)
③ Poda (MinSupport, k)
```

① Se realiza un recorrido secuencial de la tabla de datos.

② Para cada tupla *t* de la tabla *Datos*, se mira qué k-itemsets candidatos contiene y se incrementan los contadores asociados a cada uno de ellos. Este proceso sirve para calcular la relevancia de los itemsets candidatos.

③ Finalmente, se eliminan de la estructura de datos todos aquellos k-itemsets cuya relevancia sea inferior al umbral preestablecido *MinSupport*.

Para realizar su misión, la primitiva RELEVANTES(K) llama a dos rutinas auxliares COUNT(T,K) y PODA(MINSUPPORT,K), encargadas de contabilizar los k-itemsets de las tuplas y eliminar los k-itemsets no relevantes del árbol de itemsets, respectivamente.

Rutina auxiliar: COUNT(T,K)

Para contabilizar los itemsets incluidos en una tupla de una forma eficiente utilizamos la tabla hash que se encuentra en la raíz del árbol de itemsets, tal como queda descrito en el siguiente fragmento de pseudocódigo:

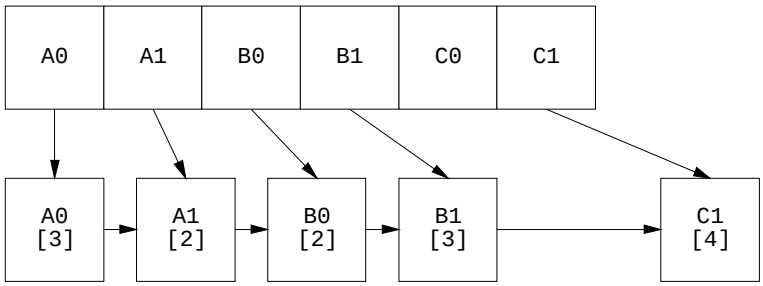
```
Si k=1
   Para cada dominio v de cada atributo a de la tupla t
        hash[base[a]+v]++
si no
   Para cada dominio v de cada atributo a de la tupla t
        Si hash[base[a]+v]!=null
           hash[base[a]+v].Count (t,k-1)
```

Rutina auxiliar: PODA(MINSUPPORT,K)

Para eliminar del árbol de itemsets aquéllos k-itemsets que no alcanzan la relevancia mínima basta una llamada al método que realiza la poda del subárbol que cuelga de un nodo. Obviamente la llamada debe realizarse sobre el nodo raíz:

```
root.Poda(MinSupport)

Si k=1
   Actualizar tabla hash (algunas entradas habrá que anularlas)
```

La única peculiaridad de la rutina consiste en que hay que actualizar la tabla hash (eliminar punteros a nodos que ya no son del árbol) al podar el conjunto de candidatos $C[1]$:



Obsérvese cómo se ha eliminado la referencia al ítem $C:0$ en la tabla hash del árbol de itemsets correspondiente a la tabla de ejemplo.

PRIMITIVAS DEL ÁRBOL DE ITEMSETS: CANDIDATOS (K)

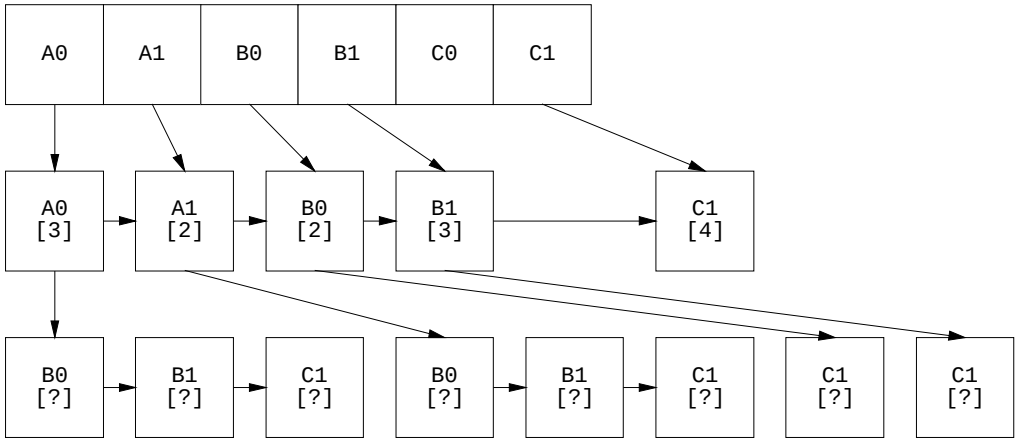
Esta primitiva es la que se encarga de generar los k-itemsets candidatos (el conjunto *C[k]*) a partir de los (k-1)-itemsets contenidos en el árbol, que corresponden al conjunto de itemsets relevantes *L[k-1]*.

Esta rutina consiste simple y llanamente en llamar al correspondiente método del nodo raíz del árbol (el primer ítem de *L[1]*):

```
root.Candidatos(k)
```

vg:

El resultado de llamar a la primitiva para la generación del conjunto de candidatos C[2] correspondiente a la tabla de ejemplo es el siguiente:



Como se puede comprobar con facilidad, la generación de candidatos consiste simplemente en poner como hijos de un nodo todos sus hermanos a la derecha excepto aquéllos que son del mismo atributo (dado que un itemset no puede tener dos items correspondientes a la misma columna de la tabla).

PRIMITIVAS DEL ÁRBOL DE ITEMSETS: REGLAS (MINCONFIDENCE)

La última primitiva del TDA "Árbol de Itemsets" obtiene todas las reglas de asociación que se puedan derivar a partir de los k-itemsets incluidos en la estructura de datos (aquéllas cuya fiabilidad alcance *MinConfidence*).

Para ello utiliza un TDA auxiliar, denominado *KItemset*, que incluye dos iteradores necesarios para recorrer el árbol de itemsets de la forma adecuada para la obtención de las reglas de asociación. Uno de esos iteradores [*nextRuleItemset*] va obteniendo k-itemsets relevantes con $k \geq 2$ (aquéllos de los que se puede derivar una regla) y el otro [*nextSubItemset*] se encarga de ir devolviendo subconjuntos propios de un itemset dado.

El proceso de obtención de reglas es el mismo de siempre:

```
Para cada itemset relevante l_k = {attr1:val1 .. attrK:valK} ∈ L[k], k≥2
     // Generar reglas a partir de l_k
     Para cada ítem l_i ⊂ l_k
          Si support(l_k)/support(l_i) ≥ MinConfidence
             Regla l_i ⇒ (l_k-l_i)  [support(l_k)/support(l_i)]
```

En un lenguaje de alto nivel típico el algoritmo quedaría más o menos así:

```
tmp = KItemset (root)
while (tmp.nextRuleItemset)
    for (i=0; i<tmp.k; i++)
        aux = KItemset ( hash[base[tmp.Atributo(i)] + tmp.Valor(i) ] )
        while (aux!=null)
              if (tmp.Support()>=MinConfidence*aux.Support())
                   Regla: aux -> (tmp-aux) [tmp.Support/aux.Support]
              aux = tmp.nextSubItemset (aux);
```

Rutinas empleadas en las primitivas del árbol de itemsets

Las primitivas del árbol de itemsets hacen uso de una serie de rutinas auxiliares que, si estuviésemos programando en un lenguaje orientado a objetos corresponderían a dos clases auxiliares:

CLASE NODOITEM

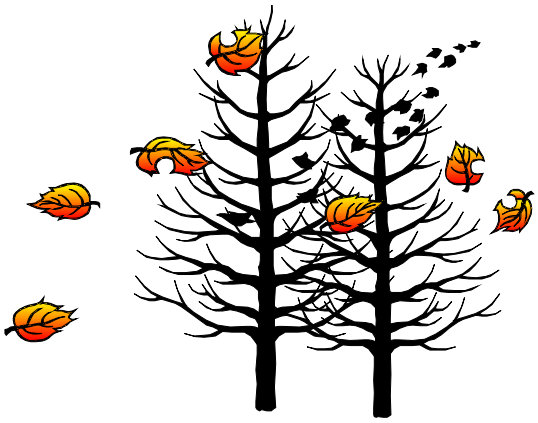
Rutinas para manejar independientemente los nodos del árbol de itemsets:

- *Count (t, k)*
- *Poda (MinSupport)*
- *Candidatos (k)*
- *Items(k)*

CLASE KITEMSET

Iteradores necesarios para generar las reglas de asociación:

- *nextRuleItemset*
- *nextSubItemset (prev)*



NodoItem

A continuación se describe sucintamente el contenido de la clase auxiliar NodoItem, sus variables de instancia y sus métodos (esenciales para el algoritmo TBAR):

Variables de instancia

```
int      Atributo;            // ID columna

int      Valor;               // ID dominio

int      Count;               // Support

NodoItem Alternativa;         // Hermano a la derecha

NodoItem Continuación;        // Hijo más a la izquierda
```

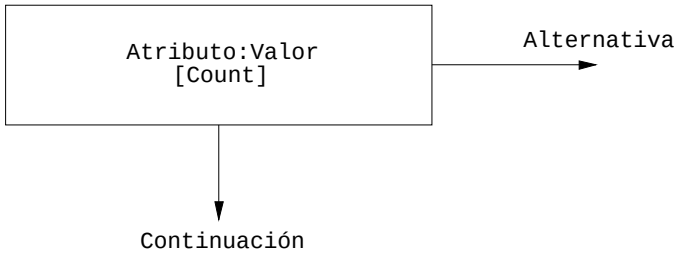


Figura: La representación de un nodo del árbol de itemsets

Método COUNT (T, K)

```
si Atributo:Valor ∈ t

   Continuación.Count(t,k-1)

   si (k=1)
      Count++

Alternativa.Count(t,k)
```

Método PODA (MINSUPPORT)

```
Mientras Continuación.Count < MinSupport
      Continuación = Continuación.Alternativa

Continuación.Poda (MinSupport)

Mientras Alternativa.Count < MinSupport
      Alternativa = Alternativa.Alternativa

Alternativa.Poda (MinSupport)
```

Método CANDIDATOS (K)

```
Si k=2

      Generar los candidatos derivados del nodo

      Alternativa.Candidatos (k)

si no

      Continuación.Candidatos (k-1)
```

Nota acerca de la generación de candidatos: Los candidatos derivados de un nodo se obtienen simplemente copiando la lista de hermanos a la derecha del nodo (sin incluir aquellos cuyo atributo coincide con el del nodo) y poniéndolos a continuación como hijos del nodo.

Método ITEMS(K)

```
total = 0

Si k=1
      total = 1
si no
      total = Continuación.Items (k-1);

total += Alternativa.Items (k);

return total;
```

Kitemset

Kitemset es una clase auxiliar que encapsula un k-itemset (en realidad se puede implementar como un array de referencias a nodos del árbol de itemsets) y permite recorrer adecuadamente el árbol de itemsets en el proceso de obtención de reglas de asociación a partir de los conjuntos de itemsets relevantes *L[k]*:

Iterador NEXTRULEITEMSET

NEXTRULEITEMSET devuelve el *siguiente k-itemset (k≥2)* contenido en la estructura de datos utilizada por el algoritmo *TBAR* (el árbol de itemsets). Este iterador equivale a la realización del recorrido secuencial por las tablas *L[k]* con *k>1* que se realiza durante la generación de ***reglas*** en los algoritmos *T* y *t*.

El iterador primero intenta conseguir un (k+1)-itemset que incluya al k-itemset actual (siguiendo el enlace *Continuación* del nodo correspondiente al k-ésimo item).

Si no lo consigue, busca k-itemsets alternativos (siguiendo el enlace *Alternativa* del nodo correspondiente al último ítem del itemset).

Si tampoco lo lograse, realiza una vuelta atrás (un proceso de *backtracking*) para encontrar algún *m*-itemset (*m<k*) que se derive de los m primeros items del k-itemset actual: *m-1* items idénticos y el último encontrado a través del enlace *Alternativa*. Si sólo se consigue un 1-itemset, se procura completar éste para obtener un 2-itemset.

Iterador NEXTSUBITEMSET (PREV)

Este iterador devuelve ***subconjuntos propios*** del itemset actual, continuando el recorrido por la estructura de datos a partir del itemset *prev* (que está contenido en el actual).

Primero se buscan (en profundidad) subconjuntos del itemset actual que a su vez incluyan a *prev* (siguiendo el enlace *Continuación*). Si no se consigue, se realiza un proceso de vuelta atrás como en el iterador anterior para encontrar itemsets alternativos que estén incluidos en el itemset actual.

4.3 Refinamientos del algoritmo

ELIMINACIÓN DE CÁLCULOS REDUNDANTES

Si analizamos detenidamente la etapa de generación de los itemsets relevantes podemos apreciar cómo cada comprobación del número de k-itemsets almacenados en el árbol de itemsets implica un recorrido por la estructura de datos completa que se podría suprimir con facilidad si conforme vamos obteniendo los k-itemsets (relevantes o candidatos) contabilizamos su número:

```
① set.Init (MinSupport)
② set.Relevantes(1);
③ for (k=2; k<=Datos.columnas && set.itemsets(k-1)>=k; k++)
④      set.Candidatos(k);
⑤      if (set.itemsets(k)>0)
⑥          set.Relevantes(k);
```

Generación original de itemsets relevantes

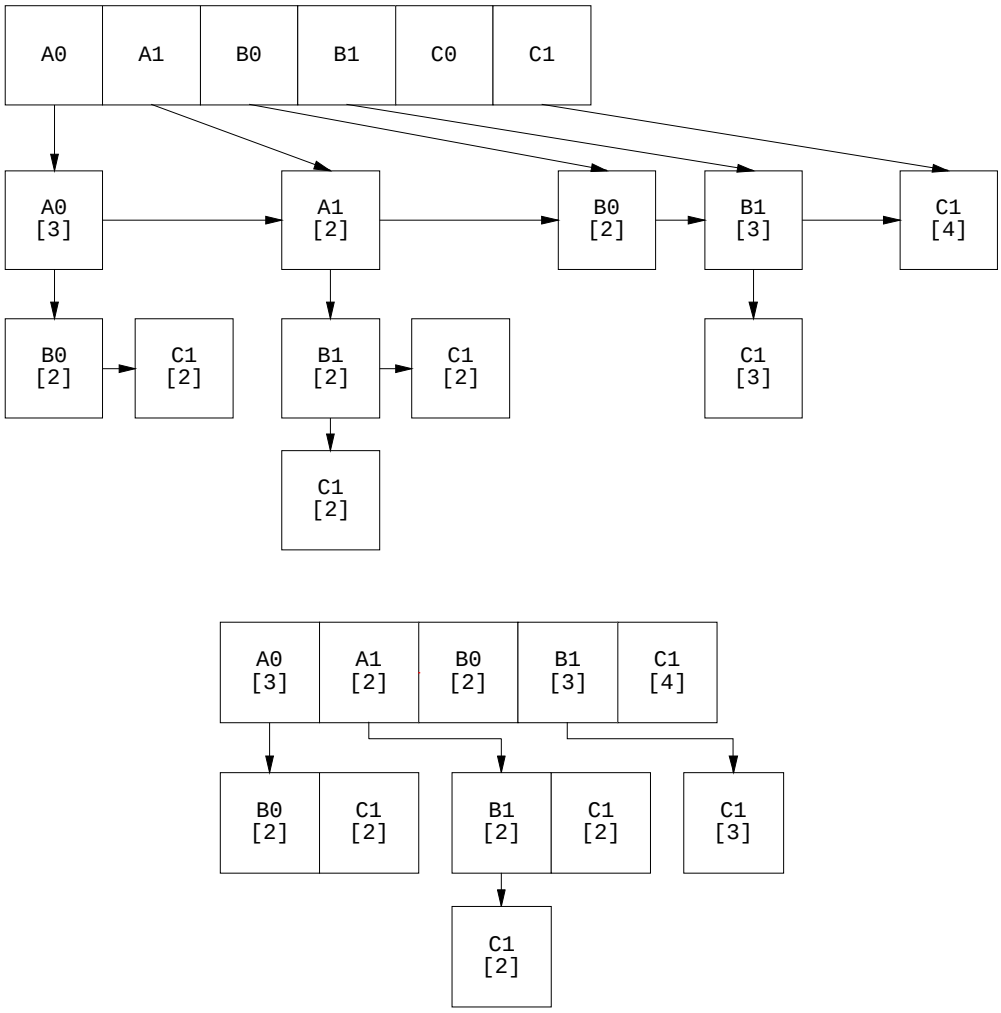
```
① set.Init (MinSupport)
② itemsets = set.Relevantes(1);
③ for (k=2; k<=Datos.columnas && itemsets>=k; k++)
④      itemsets = set.Candidatos(k);
⑤      if (itemsets>0)
⑥          itemsets = set.Relevantes(k);
```

Generación mejorada de itemsets relevantes

HASHING
Una perspectiva alternativa

El árbol de itemsets puede interpretarse desde un punto de vista diferente. Todos los items relevantes *L[1]* pueden interpretarse como pertenecientes a la raíz del árbol. A cada ítem que tenga extensiones en *L[2]* (es decir, sea comienzo de itemsets en *L[2]*) le corresponderá un hijo del nodo raíz. Análogamente, un nodo correspondiente a un itemset de *L[k]* tendrá tantos nodos hijo como extensiones del itemset pertenezcan a *L[k+1]*.

De esta forma, la estructura de datos en árbol que determina los conjuntos de itemsets se caracteriza por tener una mayor homogeneidad, sin distinción de si un nodo es raíz o no del árbol:



El árbol de itemsets tal como se veía hasta ahora (arriba)
y una visión más homogénea [y compacta] del mismo (abajo)

Observando el árbol desde esta perspectiva diferente se puede apreciar cómo, al igual que antes creábamos una tabla hash en la raíz del árbol, podemos asociar tablas hash internas a cada uno de los nodos.

Inicialmente el árbol de itemsets estaría vacío. Al ir introduciendo los items candidatos (el conjunto $C[1]$) en la estructura de datos, automáticamente se crearía una tabla hash tal como la descrita en la exposición de la versión básica del algoritmo TBAR.

Dado un nodo cualquiera, al generar los candidatos que cuelgan de un itemset incluido en el nodo (copiando los itemsets a la derecha del itemset del cual estamos derivando candidatos), podríamos crear una tabla hash si el número de itemsets del nodo recién creado supera cierto umbral preestablecido.

Hashing básico

Si el número de itemsets de un nodo supera un umbral T_1 podríamos crear una tabla hash indexada por el atributo a del ítem $a:v$. La función hash sería de la forma:

$$h_1(a:v) = h[a - a_{min}]$$

donde a_{min} corresponde al primer atributo que contiene items en el nodo actual.

Hashing avanzado

Si en determinado momento el número de itemsets de un nodo superase un umbral T_2, siendo $T_2 > T_1$, se podría crear en el nodo una tabla hash perfecta tal como la que se crea siempre en la raíz del árbol de itemsets. La función hash resultante sería de la forma:

$$h_2(a:v) = h[base[a-a_{min}]+v]$$

DHP [DIRECT HASHING AND PRUNNING]

Cuantos más itemsets se incluyan en el conjunto de candidatos más tiempo empleará el algoritmo *TBAR* (y cualquier otro) en encontrar los itemsets relevantes. La idea básica introducida por el algoritmo *DHP* [*Park, Chen & Yu: "An Effective Hash-Based Algorithm for Mining Association Rules", ACM SIGMOD'95*] es la utilización de una heurística que permita generar únicamente aquellos itemsets candidatos con una probabilidad elevada de ser relevantes.

El algoritmo *DHP*, otro derivado de *Apriori*, procura reducir el número de candidatos generados utilizando una tabla hash para (k+1)-itemsets al generar *L[k]*. Esta tabla hash ha de utilizarse sobre todo en las primeras iteraciones (que son las que producen mayor número de candidatos).

Cuando tengamos un itemset *i* perteneciente a *L[k]* incrementaremos los contadores de las entradas de la tabla hash *h[c]* para todos los (k+1)-itemsets candidatos *c* que se puedan derivar del itemset *i*. Cuando vayamos a generar *C[k+1]*, dado un candidato *c*, si *h[c]* es menor que el umbral preestablecido (<*MinSupport*) automáticamente descartaremos ese candidato: no lo incluiremos en *C[k+1]* ya que sabemos que no podrá pertenecer a *L[k+1]*.

La adaptación de la técnica descrita al caso particular de las bases de datos relacionales es directa. Al realizar la primera pasada por la base de datos completa (para obtener *L[1]*) iremos contabilizando las apariciones de itemsets en las entradas correspondientes de la tabla hash:

Poda del conjunto de candidatos C[2]

Dada una tupla, incrementamos los contadores asociados a las entradas de la tabla hash para todos y cada uno de los itemsets de 2 elementos incluidos en la tupla.

Una función hash adecuada puede construirse de una forma similar a la función hash usada en la raíz del árbol de itemsets:

$$h_{C[2]}(a_1{:}v_1,a_2{:}v_2) = ((base[a_1]+v_1)*d + base[a_2]+v_2) \bmod N$$

donde *N* indica el tamaño de la tabla hash empleada y *d* es el número total de dominios existentes en la tabla (el número de items *a:v* diferentes).

La función hash descrita se puede definir recursivamente de la siguiente manera:

$$h_{C[1]}(a{:}v) = base[a] + v$$

$$h_{C[k]}(a_1{:}v_1 \,..\, a_K{:}v_K) = (h_{C[k-1]}(a_1{:}v_1 \,..\, a_{K-1}{:}v_{K-1})*d + h_{C[1]}(a_K{:}v_K)) \bmod N$$

Obviamente, para que el conjunto de candidatos *C[2]* generado sea óptimo (igual al conjunto de itemsets relevantes *L[2]*) se necesita una función hash perfecta. Para conseguirlo, la tabla hash debería tener d^2 entradas. Esto equivale a un array bidimensional para contabilizar los 2-itemsets a la vez que se obtiene el conjunto *L[1]*, que es la técnica utilizada por Agrawal y Skirant para obtener eficientemente *L[2]*. No obstante, disponer de un array bidimensional así es inviable en cuanto el número de items diferentes es elevado.

El uso de *DHP* es especialmente útil en la generación del conjunto de candidatos *C[2]*, donde el algoritmo *TBAR* básico incluía $\binom{\#L[1]}{2}$ itemsets. Cuando $\#L[1]$ es elevado, *C[2]* es un conjunto enorme de itemsets y conviene reducir su tamaño al máximo utilizando *DHP*.

Poda del conjunto de candidatos C[3]

Igual que se podaba el conjunto de candidatos *C[2]*, la misma técnica puede emplearse para reducir el tamaño del conjunto de 3-itemsets candidatos. Simplemente tendremos que mantener otra tabla hash para la generación del conjunto *C[3]* (que no tiene por qué ser del mismo tamaño que la usada para construir *C[2]*).

En este caso, para que el conjunto de candidatos *C[3]* generado sea óptimo (idéntico al conjunto *L[3]*) se necesitaría una función hash perfecta, para lo cual la tabla hash debería tener d^3 entradas (un tamaño excesivo incluso para valores pequeños de *d*).

El uso de DHP se podría seguir extendiendo para reducir el tamaño de los conjuntos de candidatos de órdenes superiores, pero generalmente no compensará el esfuerzo realizado inicialmente para la construcción de las tablas hash con el ahorro conseguido en la generación del conjunto de candidatos. Hay que tener en cuenta que de una tupla de *n* atributos se pueden derivar como mínimo $\binom{n}{k}$ k-itemsets candidatos (suponiendo que cada valor de cada atributo únicamente pertenece a un dominio de ese atributo).

PREPROCESAMIENTO

A la hora de aplicar un algoritmo de extracción de reglas de asociación a una tabla generalmente se introducirán etapas previas de preprocesamiento. Este preprocesamiento puede consistir simplemente en obtener los valores diferentes de cada uno de los atributos, en agrupar valores en dominios o en discretizar atributos numéricos en intervalos.

Ya que el análisis de los dominios de los atributos siempre se realizará antes de aplicar el algoritmo TBAR (aunque sea sólo para establecer la codificación de los valores de los atributos), no supone ningún esfuerzo adicional obtener la relevancia de los distintos items conforme se va recopilando información de los atributos.

Al realizar el necesario recorrido por la tabla para la determinación de los dominios de los atributos (paso previo al algoritmo *TBAR* en sí) se puede contabilizar el número de ocurrencias de cada ítem, con lo cual nos ahorramos tener que recorrer la base de datos para obtener el conjunto de itemsets relevantes *L[1]*.

```
① set.Init (MinSupport)
② itemsets = set.Relevantes(1);
③ for (k=2; k<=Datos.columnas && itemsets>=k; k++)
④      itemsets = set.Candidatos(k);
⑤      if (itemsets>0)
⑥          itemsets = set.Relevantes(k);
```

Generación habitual del conjunto de itemsets relevantes

```
① set.Init (MinSupport, Datos.Dominios)
③ for (k=2; k<=Datos.columnas && itemsets>=k; k++)
④      itemsets = set.Candidatos(k);
⑤      if (itemsets>0)
⑥          itemsets = set.Relevantes(k);
```

Generación del conjunto de itemsets relevantes
aprovechando la información recopilada acerca de los dominios de los atributos

4.4 El algoritmo paso a paso

Usemos por última vez la tabla de ejemplo de la entidad de crédito, fijando una vez más el parámetro *MinSupport* en el 40% (es decir, 2 tuplas):

Aval	Nómina	Crédito
No	0	0
No	10.000	25.000
No	75.000	500.000
Sí	250.000	2.000.000
Sí	1.000.000	20.000.000

Tabla de datos original

Para esta tabla de datos, el algoritmo *TBAR* podría utilizar la siguiente información acerca de los atributos (de un modo análogo a como lo hacía el algoritmo *T*):

ID	ATTR	TIPO
0 (A)	Aval	CHAR
1 (B)	Nómina	NUMBER
2 (C)	Crédito	NUMBER

Tabla *ATTR*

Aunque *TBAR* utiliza enteros para codificar tanto atributos como dominios, en el desarrollo del ejemplo se emplearán las primeras letras mayúsculas del alfabeto a la hora de especificar el atributo de los items (simplemente para mejorar la legibilidad en la exposición).

Las siguientes tablas describen los dominios en los que se agrupan los valores de los tres atributos de la tabla del ejemplo. La tabla que describe los dominios del atributo "Aval" es trivial y no necesita comentarios. Los otros dos atributos, al ser numéricos, se discretizan en intervalos.

ID	VAL
0	No
1	Sí

Dominios del atributo A: "Aval"

ID	INF	SUP
0	0	49.999
1	50.000	100.000.000

Dominios del atributo B: "Nómina"

ID	INF	SUP
0	0	0
1	1	100.000.000

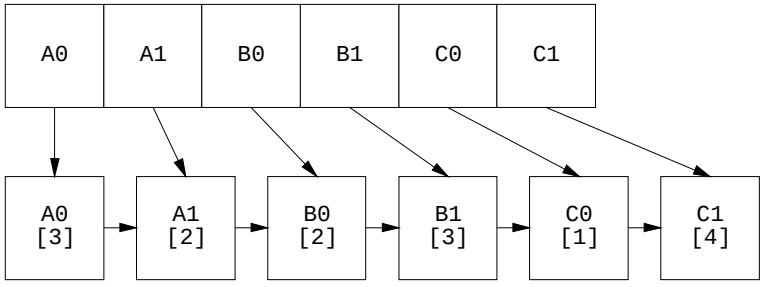
Dominios del atributo C: "Crédito"

Con la información disponible acerca de los atributos de la tabla y sus dominios, la tabla de datos original queda codificada como se muestra a continuación:

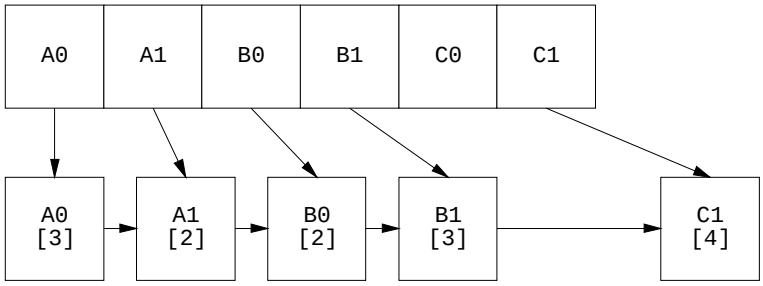
A	B	C
0	0	0
0	0	1
0	1	1
1	1	1
1	1	1

Tabla de datos codificada

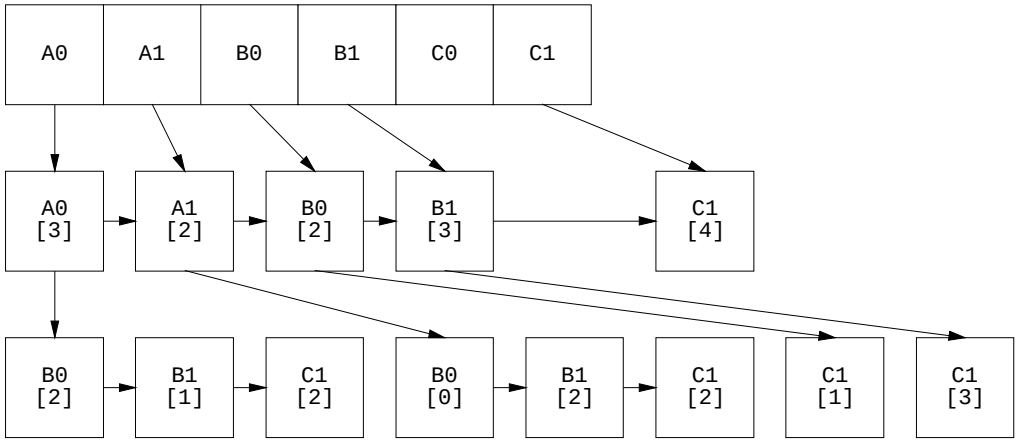
Generación del conjunto de candidatos C[1]



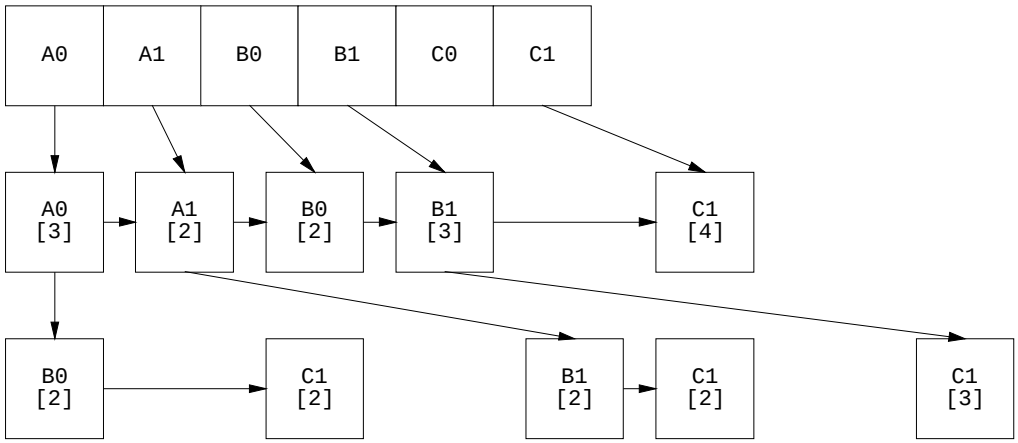
Obtención del conjunto de itemsets relevantes L[1]



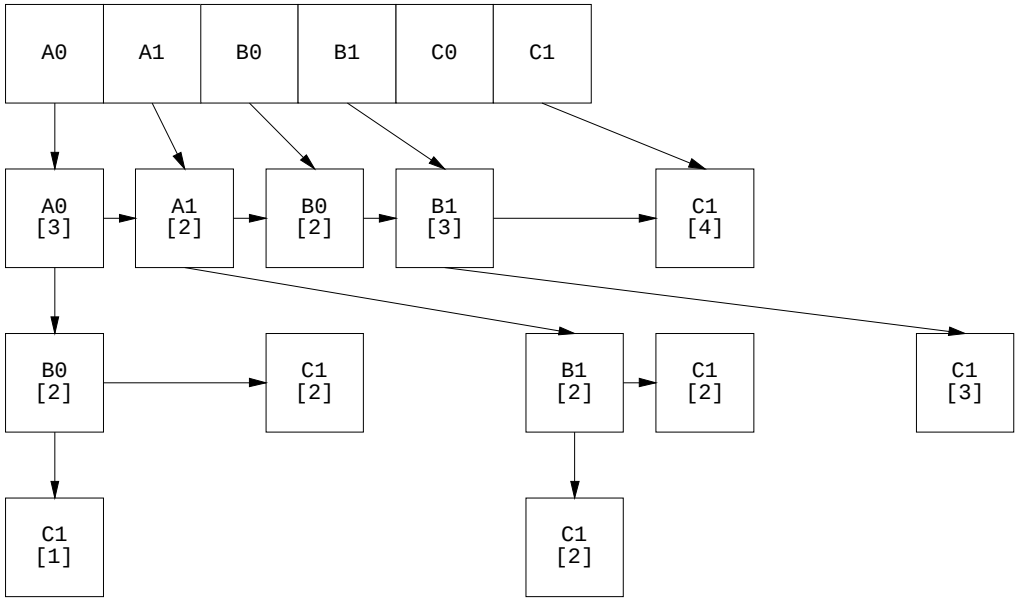
Generación del conjunto de candidatos C[2]



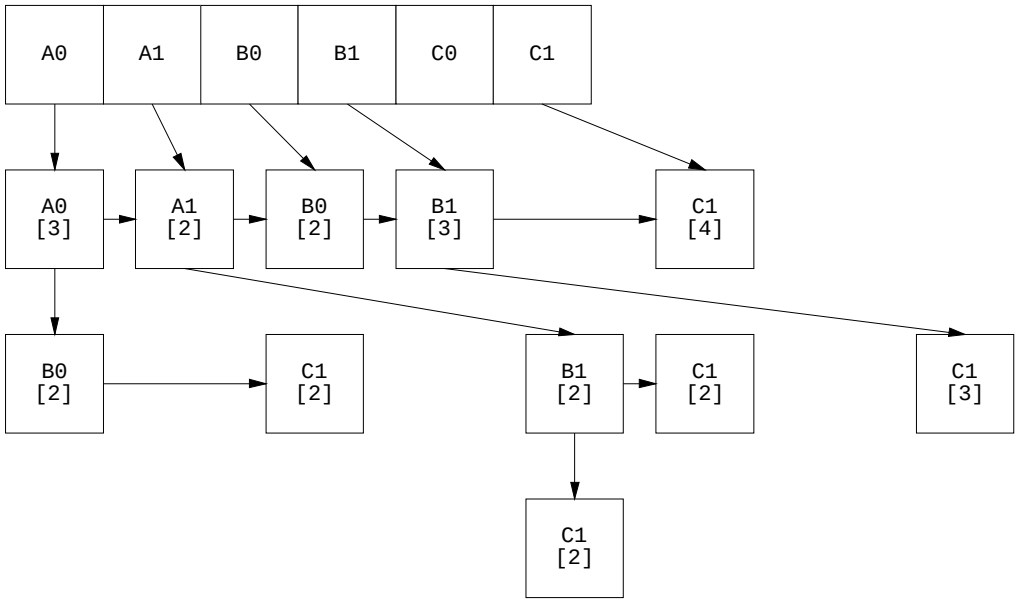
Itemsets relevantes L[1]∪L[2]



Generación del conjunto de candidatos C[3]



Conjunto final de itemsets relevantes (L[1]∪L[2]∪L[3])



De la estructura de datos generada durante la aplicación del algoritmo TBAR a la tabla de ejemplo se obtienen con facilidad las reglas de asociación (en el orden en el que aparecen en la tabla siguiente):

Regla de asociación	Interpretación	Fiabilidad
A:0 ⇒ B:0	Aval:No ⇒ Nómina<50000	2/3 66%
B:0 ⇒ A:0	Nómina<50000 ⇒ Aval:No	2/2 100%
A:0 ⇒ C:1	Aval:No ⇒ Crédito:Sí	2/3 66%
C:1 ⇒ A:0	Crédito:Sí ⇒ Aval:No	2/4 50%
A:1 ⇒ B:1	Aval:Sí ⇒ Nómina≥50000	2/2 100%
B:1 ⇒ A:1	Nómina≥50000 ⇒ Aval:Sí	2/3 66%
A:1 ⇒ B:1 C:1	Aval:Sí ⇒ Nómina≥50000 ∧ Crédito:Sí	2/2 100%
A:1 B:1 ⇒ C:1	Aval:Sí ∧ Nómina≥50000 ⇒ Crédito:Sí	2/2 100%
A:1 C:1 ⇒ B:1	Aval:Sí ∧ Crédito:Sí ⇒ Nómina≥50000	2/2 100%
B:1 ⇒ A:1 C:1	Nómina≥50000 ⇒ Aval:Sí ∧ Crédito:Sí	2/3 66%
B:1 C:1 ⇒ A:1	Nómina≥50000 ∧ Crédito:Sí ⇒ Aval:Sí	2/3 66%
C:1 ⇒ A:1 B:1	Crédito:Sí ⇒ Aval:Sí ∧ Nómina≥50000	2/4 50%
A:1 ⇒ C:1	Aval:Sí ⇒ Crédito:Sí	2/2 100%
C:1 ⇒ A:1	Crédito:Sí ⇒ Aval:Sí	2/4 50%
B:1 ⇒ C:1	Nómina≥50000 ⇒ Crédito:Sí	3/3 100%
C:1 ⇒ B:1	Crédito:Sí ⇒ Nómina≥50000	3/4 75%

El proceso de obtención de reglas es análogo al realizado por los algoritmos *t* y *T*. Únicamente cambia el orden en que se obtienen las reglas (debido a la forma de recorrer la estructura de datos que contiene los conjuntos de itemsets relevantes).

5. Resultados experimentales

Se han probado los algoritmos *t* , *T* y *TBAR* con distintas tablas de prueba: *TEST* (de 1000 tuplas generadas aleatoriamente), *SIMPLE* (con 1024 tuplas diferentes), *EMP* (con las fichas de unos cuantos empleados) e *ITEM* (con datos de algunas transacciones comerciales).

Los algoritmos se han codificado en Java (versión 1.1 del JDK [Java Development Kit] de Sun Microsystems). Se ha utilizado Personal Oracle Lite 3.0 como servidor de bases de datos relacionales y JDBC [Java DataBase Connectivity] para el acceso a las tablas desde Java.

Los tiempos han sido medidos en un ordenador personal PC con microprocesador Intel Pentium a 166 MHz y 32 MB de memoria principal (en cuatro módulos SIMM de EDO-RAM de 60 ns). El sistema operativo sobre el que se han ejecutado los algoritmos de *Data Mining* ha sido Microsoft Windows NT 4.0 Workstation (Build 1381).

Algoritmo *t*

Es el algoritmo más sencillo de los descritos. Maneja única y exclusivamente tablas relacionales y no permite agrupar los valores de los atributos en dominios.

Se ha implementado el algoritmo *t* para poder compararlo con el algoritmo *T* (con su versión *DBMS* para ser precisos) y apreciar la pérdida de eficiencia que se produce al permitir la clasificación de los valores de los atributos en dominios. Como es lógico, la ejecución del algoritmo *T* (la generalización del algoritmo *t*) es más lenta que la del algoritmo *t*. Se produce un incremento en el tiempo de ejecución del algoritmo pero esta disminución de eficiencia es admisible teniendo en cuenta la flexibilidad del algoritmo *T* (de la que carece el algoritmo *t*).

Algoritmo *T*

De este algoritmo se han desarrollado dos versiones, denominadas *DBMS* y *LIST*. La versión *DBMS* trabaja única y exclusivamente con tablas (tal como se describe en la sección correspondiente) mientras que en la versión *LIST* los conjuntos de itemsets se almacenan en memoria principal (mediante listas).

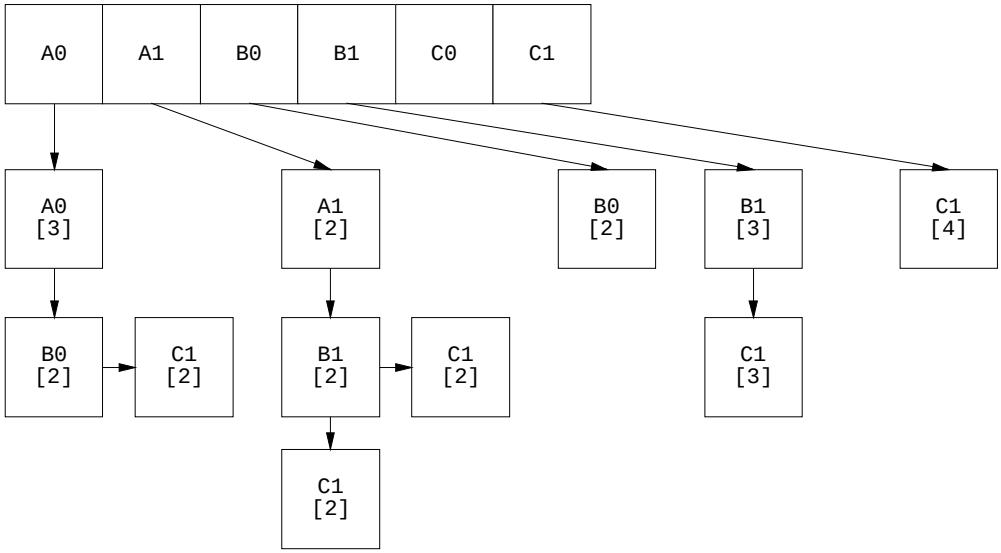
La versión *LIST* del algoritmo *T* es completamente equivalente a la versión *DBMS*. La única diferencia entre ellas es que la construcción de los conjuntos de itemsets la realiza el cliente en la versión *LIST* y el servidor en la versión *DBMS* del algoritmo.

Algoritmo *TBAR*

Del algoritmo *TBAR* [*Tree-Based Association Rule generation*], basado en la representación de todos los conjuntos de itemsets en un único árbol, se han implementado múltiples variantes:

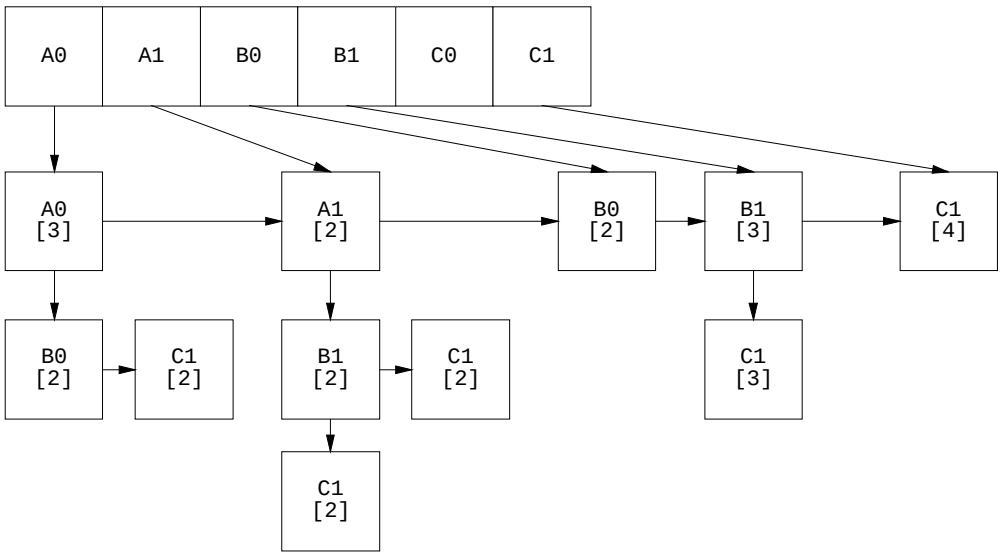
✓ *Versión HASH*

Esta versión sigue fielmente la descripción inicial del algoritmo *TBAR* realizada en la sección correspondiente a este algoritmo.



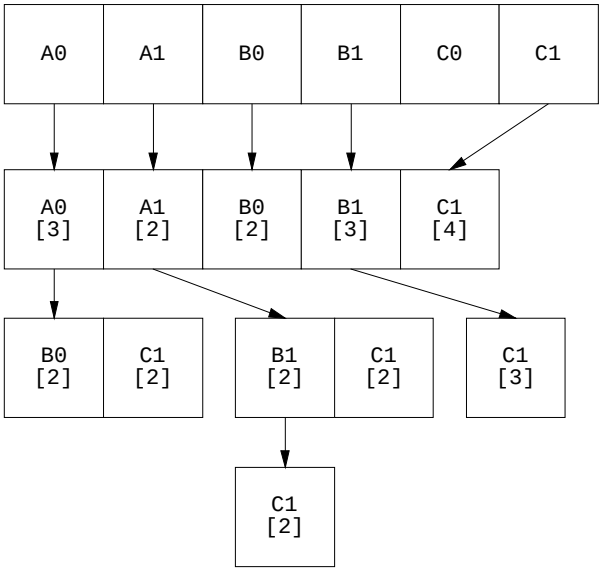
✓ *Versión HASH-LINKED*

Las versiones *HASH* y *HASH_LINKED* difieren en que en *HASH* los nodos situados a una profundidad 1 (correspondientes a los itemsets relevantes de orden 1 [los elementos de L1]) no están enlazados mientras que en *HASH_LINKED* sí lo están:



✓*Versión HASH-PACKED*

La versión *HASH_PACKED* es funcionalmente equivalente a la versión *HASH_LINKED*. En ella se empaquetan todos los hijos de un nodo en un array (para disminuir el tiempo empleado en la gestión de memoria dinámica realizada por el recolector de basura de Java):



NOTA: en la versión disponible del compilador de Java y del intérprete JVM [*Java Virtual Machine*] el tiempo necesario para crear un objeto con new crece proporcionalmente al número de objetos existentes en memoria (por lo que resulta muy caro tener una gran estructura de datos con miles de pequeños nodos).

✓*Versión DHP2*

En esta versión del algoritmo *TBAR* se incluye la poda del conjunto de candidatos *C[2]* que realiza el algoritmo *DHP* de Park, Chen y Yu. *DHP2* procura reducir el número de candidatos incluidos en *C[2]* usando una tabla hash al generar *L[1]*. En la pasada inicial por la base de datos, cuando tengamos un item *a:v* incrementaremos los contadores de las entradas de la tabla hash *h[c]* para todos los candidatos *c* derivados de *a:v* [los candidatos *(a:v,A:V)* tales que el item *A:V* también se encuentra en la tupla en la que se halla *a:v*]. Cuando vayamos a generar *C[2]*, dado un candidato *c*, si *h[c]* es menor que el umbral preestablecido (<*MinSupport*) automáticamente descartaremos ese candidato: no lo incluiremos en *C[2]* ya que sabemos que no podrá pertenecer a *L[2]*.

✓*Versión DHP3*

Esta versión de *TBAR* va un paso más allá en la aplicación de la poda de los conjuntos de candidatos. El mismo proceso que se utilizaba en *DHP2* para poder podar el conjunto *C[2]* se aplica también durante el recorrido inicial por la tabla para disminuir el tamaño de *C[3]*.

Tabla de prueba

TEST (8 columnas x 1000 tuplas)

Para comparar los distintos algoritmos implementados se ha utilizado una tabla de datos sintéticos con un millar de tuplas y ocho columnas. Todas las columnas son de tipo numérico y tienen un centenar de valores diferentes. Cada valor ha sido asociado a un dominio diferente (todos los valores pertenecen a un dominio y cada dominio incluye un único valor):

Atributo	Tipo	Número de dominios
C1	NUMBER(6,0)	100
C2	NUMBER(6,0)	100
C3	NUMBER(6,0)	100
C4	NUMBER(6,0)	100
C5	NUMBER(6,0)	100
C6	NUMBER(6,0)	100
C7	NUMBER(6,0)	100
C8	NUMBER(6,0)	100

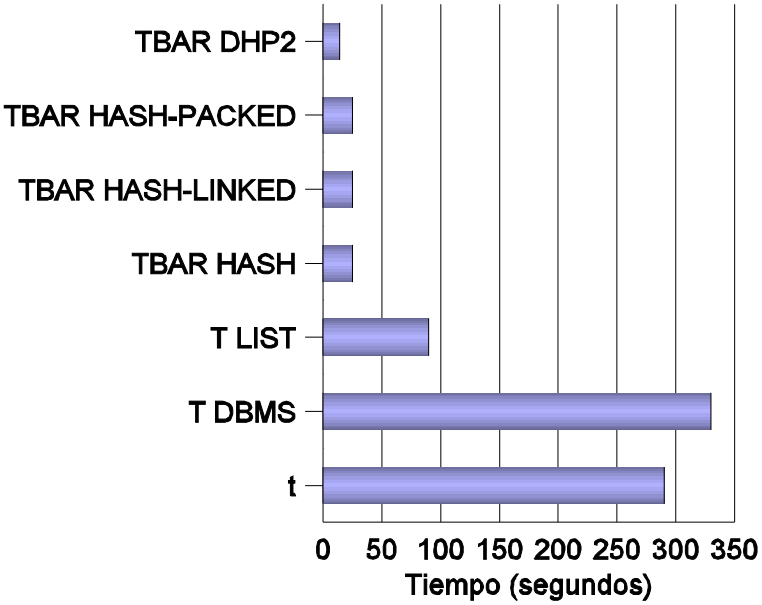
Las tuplas de la tabla se han generado de forma aleatoria utilizando un generador de números aleatorios uniforme en el intervalo [0,99].

A continuación se ofrecen los resultados obtenidos en la generación de itemsets a partir de la tabla de prueba TEST para distintos valores del parámetro *MinSupport*.

Se compararán los algoritmo *t*, *T* y *TBAR*, se analizará el comportamiento de las distintas versiones del algoritmo *TBAR* y se estudiará la influencia del tamaño de la tabla hash en el comportamiento de las versiones de *TBAR* que incorporan *DHP* (es decir, *DHP2* y *DHP3*).

MinSupport = 15 tuplas (1.5%)

Algoritmo	Versión	#L1	#C2	#L2	Tiempo
t		77	2586	0	~4'50"
T	DBMS	77	2586	0	>5'30"
	LIST	77	2586	0	1'30"
TBAR	HASH	77	2586	0	25"
	HASH-LINKED	77	2586	0	25"
	HASH-PACKED	77	2586	0	25"
	DHP2	77	0	0	14"

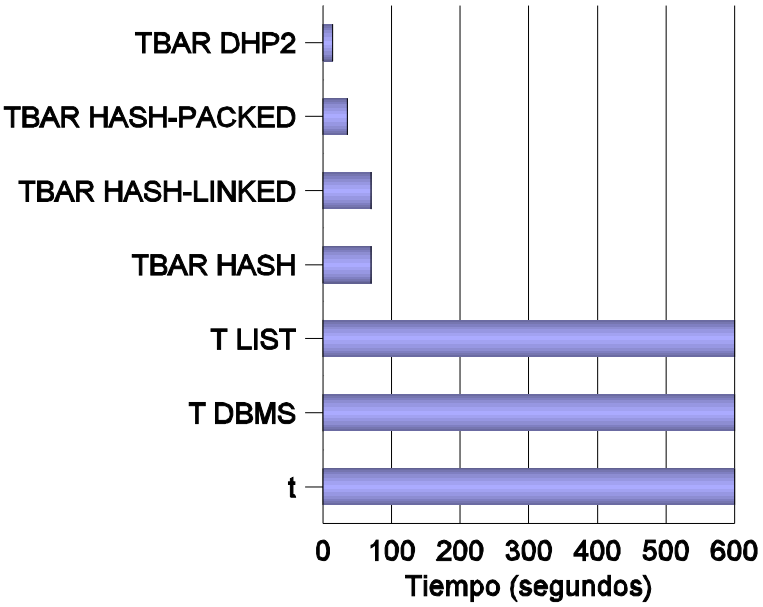


La versión DBMS del algoritmo T (aplicada a una tabla equivalente con #L1=64 y #C2=1761) realiza el recorrido inicial para obtener L1 más rápido que la versión LIST [lo hace Oracle], pero es más lento en cuanto hay que combinar tablas [Oracle no se da cuenta de que la operación se puede realizar en una sola pasada].

La versión DHP2 del algoritmo TBAR se aplicó con una tabla hash de 16384 entradas (mucho menor que el cuadrado del número de dominios existente, 640000). A pesar de no garantizar una poda óptima (tendría que tener 640000 entradas), en este caso sí la realiza.

MinSupport = 10 tuplas (1%)

Versión del algoritmo TBAR	#L1	#C2	#L2	Tiempo
HASH	442	85430	0	1'10"
HASH LINKED	442	85430	0	1'10"
HASH PACKED	442	85430	0	36"
DHP2	442	0	0	14"



Comentarios

Para esta tabla, con el umbral *MinSupport* fijado en 10 tuplas, los algoritmos *t* y *T* (tanto en su versión DBMS como en su versión LIST) consumen más de 10 minutos, un tiempo de ejecución un orden de magnitud superior al del algoritmo *TBAR*. Si además utilizamos *DHP* [*Direct Hashing and Prunning*] en el algoritmo *TBAR*, la mejora alcanza dos órdenes de magnitud.

La versión DHP2 del algoritmo TBAR se aplicó con una tabla hash de 16384 entradas (mucho menor que el cuadrado del número de dominios existente, 640000). A pesar de no garantizar una poda óptima (tendría que tener 640000 entradas), sigue realizándola.

El tamaño de la tabla hash usada en la versión DHP2 del algoritmo se mantiene en 16K entradas. Aunque se deja de obtener una poda óptima del conjunto *C[2]* en cuanto MinSupport baja de 10 tuplas, la poda obtenida sigue siendo muy buena.

MinSupport = 9 tuplas (0.9%)

Versión del algoritmo TBAR	#L1	#C2	#L2	Tiempo
HASH	535	125202	0	1'55"
HASH LINKED	535	125202	0	1'55"
HASH PACKED	535	125202	0	45"
DHP2	535	9	0	25"

MinSupport = 8 tuplas (0.8%)

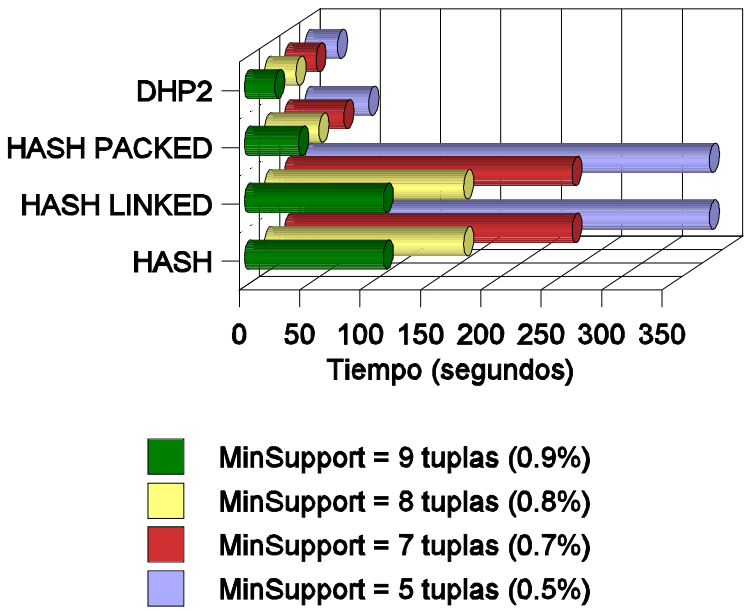
Versión del algoritmo TBAR	#L1	#C2	#L2	Tiempo
HASH	612	163822	0	2'45"
HASH LINKED	612	163822	0	2'45"
HASH PACKED	612	163822	0	45"
DHP2	612	66	0	26"

MinSupport = 7 tuplas (0.7%)

Versión del algoritmo TBAR	#L1	#C2	#L2	Tiempo
HASH	683	204075	0	3'58"
HASH LINKED	683	204075	0	3'58"
HASH PACKED	683	204075	0	49"
DHP2	683	370	0	26"

MinSupport = 5 tuplas (0.5%)

Versión del algoritmo TBAR	#L1	#C2	#L2	Tiempo
HASH	758	251366	0	5'35"
HASH LINKED	758	251366	0	5'35"
HASH PACKED	758	251366	0	53"
DHP2	758	8358	0	27"



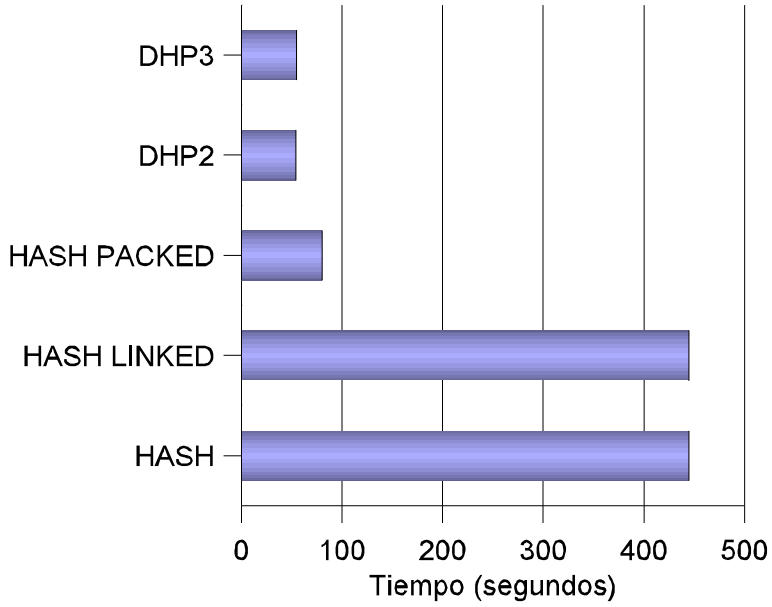
GENERACIÓN DE ITEMSETS
MinSupport = 2 tuplas (0.2%)

```
#L1 =    800
#C2 = 280000
#L2 =   1297
#C3 =   1812
#L3 =     27
#C4 =      0
```

REGLAS DE ASOCIACIÓN
MinConfidence = 0.01 (1%)

TOTAL: 2756 reglas

Versión del algoritmo TBAR	Tiempo Itemsets	Tiempo Reglas
HASH	7'00"	0'25"
HASH LINKED	7'00"	0'25"
HASH PACKED	1'10"	0'10"
DHP2	≤ 54"	
DHP3	≤ 55"	



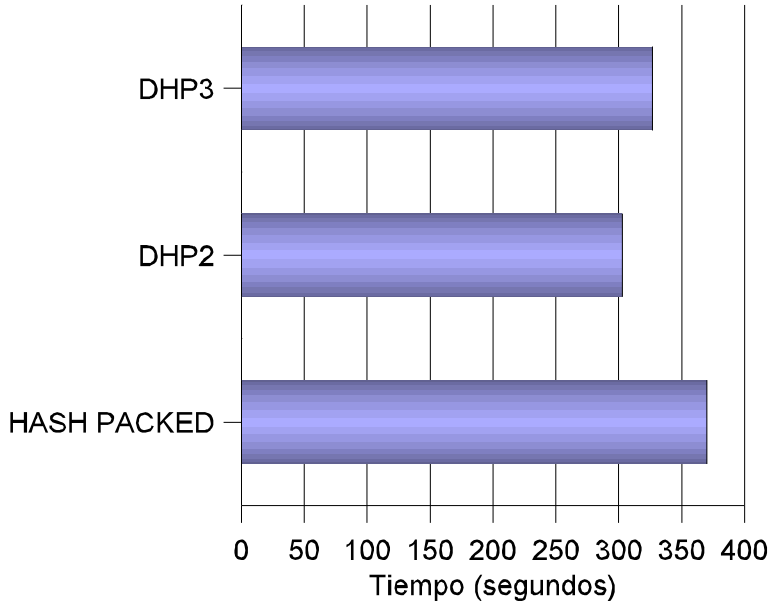
GENERACIÓN DE ITEMSETS
MinSupport = 1 tupla (0.1%)

```
#L1 =    800
#C2 = 280000
#L2 =  26652
#C3 = 561772
#L3 =  55973
#C4 =  77140
#L4 =  70000
#C5 =  56020
#L5 =  56000
#C6 =  28000
#L6 =  28000
#C7 =   8000
#L7 =   8000
#C8 =   1000
#L8 =   1000
```

REGLAS DE ASOCIACIÓN
MinConfidence = 100%

TOTAL: 4.865.763 reglas

Versión del algoritmo TBAR	Tiempo Itemsets	Tiempo Reglas
HASH PACKED	6'10"	~12'
DHP2	≤5'03"	
DHP3	≤5'27"	



ANOTACIONES:

Versiones HASH y HASH-LINKED

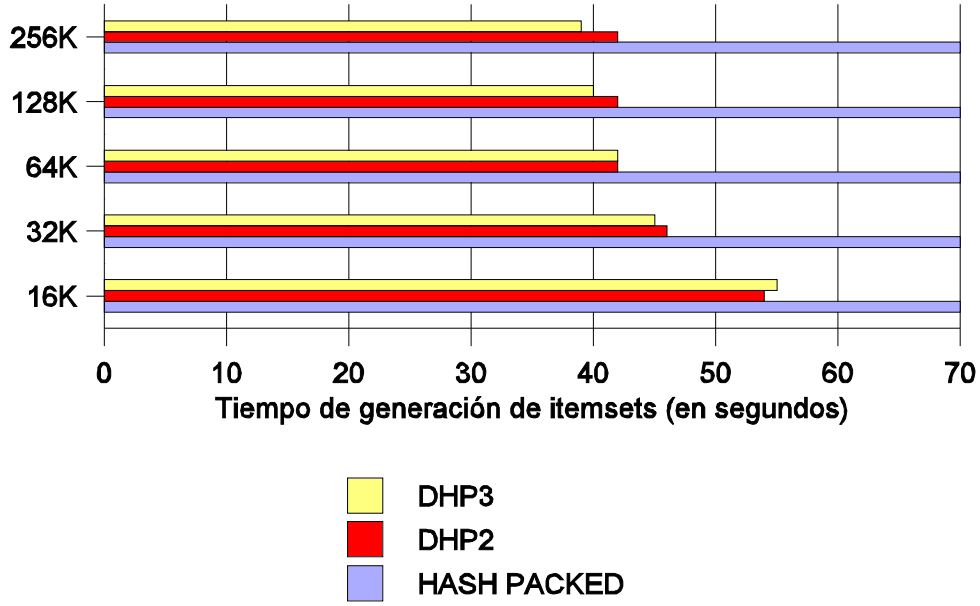
Las versiones HASH y HASH-LINKED tardan más de 7 minutos en obtener hasta L2 (y consumen más memoria). De los seis minutos que tardan en generar C2, más de cinco y medio se consumen creando nuevos objetos en memoria (ejecutando "new NodoItem").

Versiones DHP2 y DHP3

En el peor caso, la versión DHP2 es mejor que la versión DHP3, aunque esto no es cierto en general. Esto se debe a que, cuando la tabla hash es pequeña, la versión DHP3 del algoritmo no realiza una poda efectiva del conjunto de candidatos C3. DHP3 realiza más trabajo inicialmente que DHP2 (al obtener L1 y calcular los valores de las distintas entradas de la tabla hash) y, posteriormente, no saca partido del esfuerzo realizado. En cuanto el tamaño de la tabla hash aumenta, DHP3 se comporta mejor que DHP2 (si bien es cierto que consume más memoria).

MinSupport = 2 tuplas (0.2%)

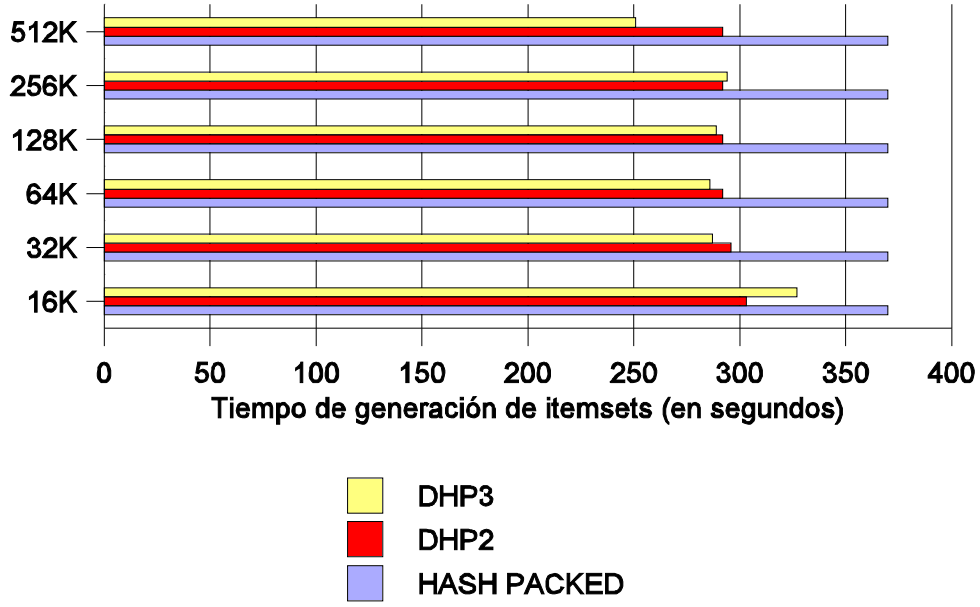
Versión	Tamaño de la tabla hash	#C2	#C3	Tiempo (itemsets)
HASH PACKED	-	280000	1812	1'10"
DHP2	16384	142609	1812	54"
	32768	60255	1812	46"
	65536	20839	1812	42"
DHP3	16384	142609	1561	55"
	32768	60255	1071	45"
	65536	20839	603	42"
	131272	11525	307	40"
	262544	3694	146	39"



MinSupport = 1 tupla (0.1%)

Versión	Tamaño de la tabla hash	#C2	#C3	Tiempo (itemsets)
HASH PACKED	-	280000	561772	6'10"
DHP2	16384	229289	561772	5'03"
	32768	160789	561772	4'56"
	65536	98481	561772	4'52"
DHP3	16384	229289	539612	5'27"
	32768	160789	464273	4'47"
	65536	98481	342863	4'46"
	131272	74161	231653	4'49"(*)
	262544	42597	153206	4'54"(*)
	524288	27431	106923	**4'11"**

(*) Durante la ejecución se agota la memoria principal y se recurre al disco [memoria virtual]



Para que la poda de *C[2]* fuese perfecta, la tabla hash debería tener N^2 entradas, siendo N el número de items diferentes. La poda perfecta de *C[3]* requeriría N^3 entradas. Para la tabla 'TEST', N es 800, N^2 llega a 640000 (bastante grande para tener una tabla de ese tamaño) y N^3 es igual a 512 millones (totalmente inviable).

Tabla SIMPLE (5 columnas x 1024 tuplas)

Esta tabla contiene una tupla de cada una de las 1024 combinaciones posibles de cinco items, cada uno de los cuales puede tener cuatro valores diferentes.

```
Support (             {a1:v1}             ) = 256
Support (          {a1:v1,a2:v2}          ) =  64
Support (       {a1:v1,a2:v2,a3:v3}       ) =  16
Support (    {a1:v1,a2:v2,a3:v3,a4:v4}    ) =   4
Support ( {a1:v1,a2:v2,a3:v3,a4:v4,a5:v5} ) =   1
```

GENERACIÓN DE ITEMSETS
MinSupport = 1 tupla

```
#C1 = #L1 =   20
#C2 = #L2 =  160
#C3 = #L3 =  640
#C4 = #L4 = 1280
#C5 = #L5 = 1024

Tiempo de generación de itemsets:  25"
```

NOTA: Obviamente, si fijamos un umbral más elevado, el número de itemsets relevantes en L[5] será 0.

OBTENCIÓN DE REGLAS DE ASOCIACIÓN

```
MinConfidence = 0.01
 TOTAL: 47680 reglas
 Tiempo total 4'53"


MinConfidence = 0.25
 TOTAL: 12140 reglas
 Tiempo total 1'36"

MinConfidence > 0.25
   TOTAL: 0 reglas
  Tiempo total ~30"
```

Todos los itemsets de *L[2]* aparecen en cuatro tuplas diferentes y los de *L[1]* en sólo una. Por lo tanto, cuando la fiabilidad mínima de las reglas se establece por encima de 1/4 no existe ningún par de itemsets que generen una regla de asociación que alcance la fiabilidad mínima.

Tabla EMP (7 columnas x 14 tuplas)

Descripción de las columnas de la tabla de empleados

Columna	Tipo	Significado
EMPNO	NUMBER(4,0)	Código del empleado
NAME	CHAR(10)	Nombre
JOB	CHAR(9)	Trabajo
MGR	NUMBER(4,0)	Jefe del empleado
SAL	NUMBER(7,2)	Sueldo
COMM	NUMBER(7,2)	Comisiones
DEPT	CHAR(10)	Departamento

Descripción de los dominios de los atributos

Columa	Dominios	#Dominios
EMPNO	1 [7369.0,7369.0] 2 [7499.0,7499.0] 3 [7521.0,7521.0] 4 [7566.0,7566.0] 5 [7654.0,7654.0] 6 [7698.0,7698.0] 7 [7782.0,7782.0] 8 [7788.0,7788.0] 9 [7839.0,7839.0] 10 [7844.0,7844.0] 11 [7876.0,7876.0] 12 [7900.0,7900.0] 13 [7902.0,7902.0] 14 [7934.0,7934.0]	14
NAME	1 ADAMS 2 ALLEN 3 BLAKE 4 CLARK 5 FORD 6 JAMES 7 JONES 8 KING 9 MARTIN 10 MILLER 11 SCOTT 12 SMITH 13 TURNER 14 WARD	14

Columa	Dominios	#Dominios
JOB	1 ANALYST 2 CLERK 3 MANAGER 4 PRESIDENT 5 SALESMAN	5
MGR	1 [7566.0,7566.0] 2 [7698.0,7698.0] 3 [7782.0,7782.0] 4 [7788.0,7788.0] 5 [7839.0,7839.0] 6 [7902.0,7902.0]	6
SAL	1 [800.0, 800.0] 2 [950.0, 950.0] 3 [1100.0,1100.0] 4 [1250.0,1250.0] 5 [1300.0,1300.0] 6 [1500.0,1500.0] 7 [1600.0,1600.0] 8 [2450.0,2450.0] 9 [2850.0,2850.0] 10 [2975.0,2975.0] 11 [3000.0,3000.0] 12 [5000.0,5000.0]	12
COMM	1 [0.0, 0.0] 2 [300.0, 300.0] 3 [500.0, 500.0] 4 [1400.0,1400.0]	4
DEPT	1 COMM 2 RD 3 ELECT	3

Tiempos empleados

Algoritmo	Tiempo total
T	8"
TBAR	< 1"

Cualquiera de las versiones implementadas del algoritmo *TBAR* resuelve el problema con un tiempo total empleado inferior a un segundo. La versión *LIST* del algoritmo *T* consigue un rendimiento similar pero la versión *DBMS* de este algoritmo consume unos ocho segundos (diez segundos si se definen las tablas de itemsets en Oracle con claves primarias [implica un tiempo de procesamiento adicional al insertar y eliminar tuplas de las tablas]).

ITEMSETS OBTENIDOS
MinSupport = 3 tuplas

```
#L[1] =  8
#C[2] = 21
#L[2] =  4
#C[3] =  1
#L[3] =  1
```

Árbol de itemsets

```
· JOB:CLERK [4]
· JOB:MANAGER [3]
      · MGR:FORD [3]
· JOB:SALESMAN [4]
      · MGR:ALLEN [4]
            · DEPT:ELECT [4]
      · DEPT:ELECT [4]
· MGR:ALLEN [5]
      · DEPT:ELECT [5]
· MGR:FORD [3]
· DEPT:COMM [3]
· DEPT:RD [5]
· DEPT:ELECT [6]
```

REGLAS DE ASOCIACIÓN DERIVADAS DEL ÁRBOL DE ITEMSETS
MinConfidence = 50%

```
       JOB:MANAGER -> MGR:FORD [3/3]
       MGR:FORD -> JOB:MANAGER [3/3]
     JOB:SALESMAN -> MGR:ALLEN [4/4]
     MGR:ALLEN -> JOB:SALESMAN [4/5]
JOB:SALESMAN -> MGR:ALLEN DEPT:ELECT [4/4]
JOB:SALESMAN MGR:ALLEN -> DEPT:ELECT [4/4]
JOB:SALESMAN DEPT:ELECT -> MGR:ALLEN [4/4]
MGR:ALLEN -> JOB:SALESMAN DEPT:ELECT [4/5]
MGR:ALLEN DEPT:ELECT -> JOB:SALESMAN [4/5]
DEPT:ELECT -> JOB:SALESMAN MGR:ALLEN [4/6]
     JOB:SALESMAN -> DEPT:ELECT [4/4]
     DEPT:ELECT -> JOB:SALESMAN [4/6]
       MGR:ALLEN -> DEPT:ELECT [5/5]
       DEPT:ELECT -> MGR:ALLEN [5/6]
```

TOTAL: 14 reglas reglas de asociación

Tabla ITEM (6 columnas x 64 tuplas)

Columna	Tipo	Dominios
ORDID	NUMBER(4,0)	21
ITEMID	NUMBER(4,0)	10
PRODID	NUMBER(6,0)	10
ACTUALPRICE	NUMBER(8,2)	23
QTY	NUMBER(8,0)	16
ITEMTOT	NUMBER(8,2)	50

GENERACIÓN DE ITEMSETS
MinSupport = 3 tuplas

```
#L1 =   51
#C2 = 1035
#L2 =   53
#C3 =   40
#L3 =   11
#C4 =    0
```

REGLAS DE ASOCIACIÓN
MinConfidence = 0.1

```
TOTAL: 172 reglas de asociación
```

Tiempos empleados en la resolución del problema

Algoritmo	Tiempo total
T	1'15"
TBAR	***~2"***

Usando la versión *DBMS* del algoritmo *T* se consume 1'15" frente al par de segundos que tarda cualquier versión del algoritmo *TBAR*. Si se utilizan claves primarias en las tablas Oracle, entonces el tiempo de ejecución del algoritmo *T* alcanza 1'20".