Capítulo 2

Reglas de asociación
[Association Rules]

1. Introducción

"Data Mining (also called Knowledge Discovery in Databases)
is the efficient discovery of previously unknown patterns in large databases"

Rakesh Agrawal & John C. Shafer: "Parallel Mining of Association Rules"
IEEE Transactions on Knowledge and Data Engineering, December 1996

"Data Mining" es un término genérico que engloba resultados de investigación, técnicas y herramientas usadas para extraer información útil de grandes bases de datos. Este proceso de extracción de información se conoce como *KDD* [*Knowledge Discovery in Databases*]. Las investigaciones en este tema incluyen análisis estadístico de datos, razonamiento basado en casos [*CBR*: *Case Based Reasoning*], técnicas de representación del conocimiento, redes neuronales y visualización de datos.

KDD se ha definido como la extracción no trivial de información potencialmente útil a partir de un gran volumen de datos en el cual la información está implícita (aunque no se conoce previamente). Se trata de interpretar grandes cantidades de datos y encontrar relaciones o patrones. Para conseguirlo harán falta técnicas de aprendizaje [*Machine Learning*], estadística y bases de datos.

Tareas comunes en *KDD* son: inducción de reglas, clasificación, clustering, reconocimiento de patrones, modelado predictivo, visualización de grandes cantidades de datos, detección de dependencias, manejo de incertidumbre... Aquí nos centraremos en la obtención de reglas de asociación y en la aplicación de las técnicas existentes al caso particular de las bases de datos relacionales.

La obtención de reglas de asociación a partir de grandes bases de datos transaccionales resulta atractiva para los estudios de marketing de las organizaciones comerciales. El uso de códigos de barras facilita la recolección de grandes cantidades de datos [*basket data*]. Sería deseable disponer de técnicas que permitan la extracción de información útil a partir del gran volumen de información almacenada en las bases de datos de transacciones.

En estas bases de datos, generalmente, una transacción contiene la fecha en la que se produjo y los items involucrados en ella (ítemes sería lo correcto en castellano). Siendo I el conjunto completo de items, una transacción es un conjunto de items $T \subseteq I$ al que se le asocia un identificador único TID. Una transacción contiene un conjunto de items X si $X \subseteq T$. A partir de aquí denominaremos *itemset* a un conjunto de items (para evitar confusiones cuando hablemos de conjuntos de itemsets).

Una regla de asociación es una implicación 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 transacciones con X que contienen también a Y. La relevancia [*support*] de la regla es la proporción de transacciones de la base de datos que contienen $X \cup Y$ (tanto X como Y).

Dado un conjunto de transacciones D, se trata de obtener todas las reglas de asociación que tengan una fiabilidad y una relevancia superiores a unos umbrales especificados por el usuario (*MinConfidence* y *MinSupport*).

Como caso particular, los algoritmos de extracción de reglas de asociación se pueden aplicar a bases de datos relacionales. Entonces un ítem será un par *(atributo,valor)* y podemos imponer la restricción adicional de que todos los items de un itemset han de corresponder a atributos diferentes (como consecuencia de la Primera Forma Normal).

Ejemplo

Supongamos que disponemos de información acerca de las transacciones que se realizan en un supermercado. Cada transacción estará identificada por un código *TID* (aquí el nombre del cliente) y contendrá un conjunto de artículos (*items*). Una muestra de la base de datos podría ser:

Cliente	Artículos
Juan	Patatas, Huevos, Bacon
María	Pan, Leche, Huevos
Luis	Pan, Leche, Patatas, Huevos
Ana	Pan, Leche

Si fijamos la relevancia mínima en un 50% (*MinSupport=0.50*) estamos indicando que, para que una regla sea considerada, al menos debe cumplirse en el 50% de las transacciones de la base de datos (2 transacciones en este caso). Al establecer un umbral mínimo para la fiabilidad de las reglas de asociación estamos descartando aquéllas reglas que no se cumplen con la frecuencia necesaria para que sean consideradas interesantes. Dada la base de datos anterior, podemos obtener, entre otras, las siguientes reglas de asociación:

N	Regla	Support	Confidence
1	Pan ⇒ Leche	75 %	100 %
2	Leche ⇒ Pan	75 %	100 %
3	Patatas ⇒ Huevos	50 %	100 %
4	Huevos ⇒ Patatas	50 %	66.6 %
5	Pan ∧ Huevos ⇒ Leche	50 %	100 %
6	Leche ⇒ Huevos ∧ Pan	50 %	66.6 %
7	Huevos ∧ Leche ⇒ Pan	50 %	100 %
8	Pan ⇒ Huevos ∧ Leche	50 %	66.6 %
9	Leche ∧ Pan ⇒ Huevos	50 %	66.6 %
10	Huevos ⇒ Leche ∧ Pan	50 %	66.6 %

De la simple base de datos de transacciones se puede extraer bastante información útil. Por ejemplo, las dos primeras reglas nos muestran cómo es habitual comprar leche cuando se va a por el pan (algo que mucha gente hace diariamente).

Las dos siguientes nos podrían indicar que los clientes del supermercado toman con relativa frecuencia huevos con patatas fritas. Estas reglas ponen de manifiesto que la fiabilidad de la regla de asociación A⇒B obviamente no tiene por qué coincidir con la fiabilidad de la regla inversa B⇒A.

Las seis reglas que vienen a continuación son resultado de combinar varios items (tres en este caso). Se puede apreciar cómo el número de reglas que se obtienen puede ser inmenso para grandes bases de datos, con lo que se crea "un problema de *Data Mining* de segundo orden". Una vez obtenidas las reglas de asociación deberemos idear métodos que nos permitan acceder con facilidad a la información que nos interese. Como es lógico, el número de reglas que se genera se puede disminuir aumentando los umbrales preestablecidos (*MinSupport* y *MinConfidence*), pero esta técnica podría ocasionar la pérdida de información útil. Por lo tanto, habrá que facilitarle al usuario la manipulación de la información obtenida tras la extracción de las reglas.

2. "Divide y vencerás"

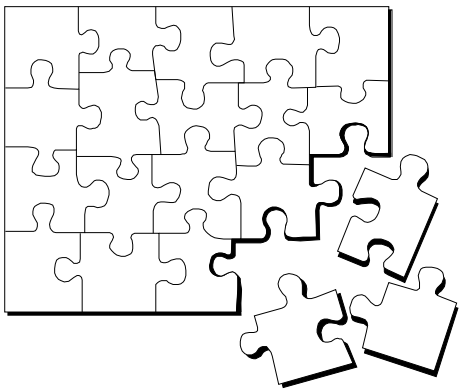
A continuación se expondrán distintos algoritmos que se han utilizado para extraer reglas de asociación de grandes bases de datos. En ellos, el problema de la obtención de todas las reglas de asociación se suele descomponer en:

① Itemsets relevantes [*covering*, *frequent* o, simplemente, *large itemsets*]

Encontrar todos los conjuntos de items [*itemsets*] con relevancia por encima de *MinSupport*. La relevancia de un conjunto de items *X* es el número de transacciones que lo contienen y se denotará *support(X)*. Los conjuntos de items cuya relevancia quede por encima del umbral serán los itemsets relevantes [*large itemsets, covering itemsets* o *frequent itemsets*] mientras que los que queden por debajo no nos interesarán [serán los "*small itemsets*"].

② Reglas de asociación [*association rules*]

Generar las reglas de asociación utilizando los conjuntos de items relevantes obtenidos en el paso anterior. Si *ABCD* y *AB* son conjuntos de items relevantes, la regla $AB \Rightarrow CD$ tendrá una fiabilidad igual al cociente *support(ABCD)/support(AB)*. La regla tendrá con seguridad una relevancia mínima (ya que *ABCD* es un conjunto relevante de items) y nos quedaremos con ella si su fiabilidad alcanza el umbral establecido [*MinConfidence*].



3. Algoritmos para la obtención de conjuntos relevantes de items

- **AIS [*Agrawal, Imielinski & Swami*]**
  Rakesh Agrawal, Tomasz Imielinsky & Arun Swami
  IBM Almaden Research Center, 1993

- **SETM [*SET-oriented Mining of association rules*]**
  Maurice Houtsma & Arun Swami
  IBM Almaden Research Center, 1993

- **Apriori, AprioriTID & AprioriHybrid**
  Rakesh Agrawal & Ramakrishnan Skirant
  IBM Almaden Research Center, 1994

- **OCD [*Offline Candidate Determination*]**
  Heikki Mannila, Hannu Toivonen & A. Inkeri Verkamo
  University of Helsinki, 1994

- **DHP [*Direct Hashing and Prunning*]**
  Jong Soo Park, Ming-Syan Chen & Philip S. Yu
  IBM Thomas J. Watson Research Center, 1995

- **Max-Miner**
  Roberto J. Bayardo Jr.
  IBM Almaden Research Center, 1998

3.1 AIS

Rakesh Agrawal, Tomasz Imielinski & Arun Swami
"Mining association rules between sets of items in large databases"
Proceedings of the ACM SIGMOD Conference on Management of Data
Washington DC, USA, May 1993

AIS es el primer algoritmo que se desarrolló para obtener reglas de asociación. De hecho, en el artículo donde aparece descrito aparece por primera vez el concepto de regla de asociación como mecanismo de representación del conocimiento simple y útil para caracterizar las regularidades que se pueden encontrar en grandes bases de datos.

En AIS, una regla de asociación es una implicación de la forma $\boldsymbol{X \Rightarrow Y\ [s,c]}$ donde X es un itemset, Y es un ítem (no incluido en X), s representa la relevancia [*support*] de la regla y c su fiabilidad [*confidence*]. En algoritmos posteriores, tanto X como Y pueden ser itemsets.

La relevancia y la fiabilidad de las reglas de asociación se suelen representar como fracciones. La fiabilidad de la regla de asociación $X \Rightarrow Y$ es la proporción de transacciones con X que contienen también a Y mientras que la relevancia de la regla es la proporción de transacciones de la base de datos que contienen $X \cup Y$.

Una de las características esenciales de las reglas de asociación es que incorporan medidas estadísticas (relevancia y fiabilidad) en las clásicas reglas de la Lógica Proposicional. De hecho, al representarlas se escogió $\Rightarrow$ para distinguirlas de las implicaciones lógicas ($\neg$).

Para evitar la generación de reglas de asociación irrelevantes, se fijan de antemano los umbrales *MinSupport* y *MinConfidence*.

Un itemset con k items será denominado ***k-itemset***. Los *k-itemsets* con una relevancia igual o superior al umbral *MinSupport* pertenecerán al conjunto ***L[k]***, el conjunto de k-itemsets relevantes (*large k-itemsets*). En el conjunto ***C[k]*** se incluirán aquéllos k-itemsets que, en principio, podrían pertenecer al conjunto *L[k]*. Los itemsets de *C[k]* se denominan k-itemsets candidatos (*candidate k-itemsets*).

Los itemsets candidatos se generan y su relevancia obtiene conforme se recorre la base de datos utilizando un contador. Hay que destacar que los itemsets se almacenan ordenados lexicográficamente.

Tras leer una transacción de la base de datos, se determinan qué itemsets relevantes de los encontrados en la pasada anterior por la base de datos están contenidos en la transacción. Los itemsets candidatos se generan extendiendo los itemsets relevantes encontrados en la transacción con otros items de la transacción.

De una transacción de N items que contiene un determinado k-itemset relevante se generan (k+1)-itemsets candidatos, los cuales se incluyen en *C[k+1]* con su contador igual a 1 (si el itemset candidato no existía anteriormente en *C[k+1]*) o ven incrementado su contador (si el itemset ya se había introducido en el conjunto de candidatos al procesar una transacción anterior).

A continuación se ofrece el esquema básico del **algoritmo AIS**:

L[k] es el conjunto de itemsets relevantes que contienen k items.

C[k] es el conjunto de k-itemsets (itemsets de k elementos) candidatos, aquéllos que pueden llegar a ser relevantes si tienen la relevancia mínima.

```
L[1] = {large 1-itemsets}
k = 2
Mientras L[k-1] ≠ ∅
      C[k] = ∅;
      Para cada transacción t ∈ D
            L_t = Conjunto de (k-1)-itemsets relevantes contenidos en t
            Para cada itemset l_t ∈ L_t
                  C_t = Extensiones de l_t contenidas en t (*)
                  Para cada candidato c ∈ C_t
                        Si c ∈ C[k]
                              Incrementar el contador asociado a c en C[k]
                        Si no
                              Añadir c a C[k] (contador=1)
      L[k] = { c ∈ C[k] | contador(c)≥MinSupport }
      k++
Solución:  ∪ L[k]
```

(*) A partir de los conjuntos relevantes de items encontrados en una transacción, se generan los candidatos expandiendo éstos con items que se encuentren en la transacción.

DETALLES DE IMPLEMENTACIÓN:

Para mantener y manejar eficientemente los conjuntos de items se utiliza un árbol hash (una tabla hash dinámica multinivel). Un nodo del árbol puede ser una lista de itemsets (hojas) o una tabla hash (nodos internos). La función hash de una tabla hash a una profundidad k se aplica al k-ésimo elemento del conjunto de items. Inicialmente, el árbol consiste en un único nodo hoja. Cuando el número de itemsets en una hoja excede un umbral prefijado, la hoja se convierte en un nodo interno. Con esta estructura de datos se puede comprobar eficientemente qué itemsets están contenidos en una transacción determinada.

EJEMPLO:

Supongamos disponemos de la base de datos del supermercado y establecemos el umbral *MinSupport* en 0.5 (para que un ítem sea relevante ha de estar incluido en, al menos, dos de las cuatro transacciones):

Cliente	Artículos
Juan	Patatas, Huevos, Bacon
María	Pan, Leche, Huevos
Luis	Pan, Leche, Patatas, Huevos
Ana	Pan, Leche

Inicialmente obtenemos el conjunto *L[1]* de items relevantes. Este conjunto está formado por los items {Patatas}[2], {Pan}[3], {Huevos}[3] y {Leche}[3]. El número que aparece entre corchetes indica la relevancia del ítem (es un contador asociado a él).

Primera iteración: Obtención de L[2]

Comenzamos el recorrido por la base de datos. De la primera transacción obtenemos los itemsets incluidos en L[1]: {Patatas} y {Huevos}. A partir de ellos generamos los candidatos {Patatas,Huevos}[1], {Patatas,Bacon}[1] y {Huevos,Bacon}[1]. Nótese que los items que incluyen a {Bacon} nunca podrán ser relevantes y, a pesar de ello, AIS los procesa (el algoritmo Apriori aprovecha esta información).

C[2]
{Patatas,Huevos} [1]
{Patatas,Bacon} [1]
{Huevos,Bacon} [1]

De la segunda transacción, que incluye tres items relevantes de *L[1]*, se obtienen los candidatos ({Pan,Leche}, {Pan,Huevos} y {Leche,Huevos}) y se actualiza el estado de *C[2]*:

C[2]
{Pan,Leche}[1]
{Pan,Huevos} [1]
{Leche,Huevos} [1]
{Patatas,Huevos} [1]
{Patatas,Bacon} [1]
{Huevos,Bacon} [1]

Continuando nuestro recorrido analizamos las transacciones realizadas por Luis y Ana, con lo que obtenemos el conjunto C[2]:

C[2]
{Pan,Leche}[3]
{Pan,Patatas} [1]
{Pan,Huevos} [2]
{Leche,Patatas} [1]
{Leche,Huevos} [2]
{Patatas,Huevos} [2]
{Patatas,Bacon} [1]
{Huevos,Bacon} [1]

Quedándonos únicamente con los itemsets candidatos cuyo contador es igual o superior al valor establecido por *MinSupport* obtenemos *L[2]*:

L[2]
{Pan,Leche}[3]
{Pan,Huevos} [2]
{Leche,Huevos} [2]
{Patatas,Huevos} [2]

Segunda iteración: Obtención de L[3]

Repitiendo el proceso anterior, esta vez con *L[2]* como entrada, vemos que la transacción realizada por Juan incluye al itemset {Patatas,Huevos} de *L[2]*, por lo que el itemset {Patatas,Huevos,Bacon} se incluye en el conjunto *C[3]*.

C[3]
{Patatas,Huevos,Bacon}[1]

De las compras de María, que incluyen el itemset {Pan,Leche} de *L[2]*, incluimos {Pan,Leche,Huevos} en *C[3]*.

C[3]
{Pan,Leche,Huevos}[1]
{Patatas,Huevos,Bacon}[1]

De la transacción realizada por Luis, que incluye varios itemsets de *L[2]*, se generan los siguientes itemsets candidatos: {Pan,Leche,Patatas}, {Pan,Leche,Huevos}, {Pan,Patatas,Huevos} y {Leche,Patatas,Huevos}.

C[3]
{Pan,Leche,Patatas}[1]
{Pan,Leche,Huevos}[2]
{Pan,Patatas,Huevos}[1]
{Leche,Patatas,Huevos}[1]
{Patatas,Huevos,Bacon}[1]

Finalmente, de la compra de Ana no se puede generar ningún 3-itemset candidato (la transacción sólo incluye dos items). Por lo tanto, el conjunto *C[3]* queda como estaba.

Para obtener *L[3]* simplemente nos quedamos con los itemsets de *C[3]* que tienen la relevancia mínima establecida por *MinSupport*:

L[3]
{Pan,Leche,Huevos}[2]

Tercera iteración: Obtención de L[4]

A partir de *L[3]* se obtiene *C[4]*, que contiene un único elemento proveniente de la transacción realizada por Luis:

C[4]
{Pan,Leche,Patatas,Huevos}[1]

Como este 4-itemset no tiene la relevancia mínima prefijada, el conjunto de itemsets *L[4]* queda vacío, por lo que no se realizan más iteraciones.

3.2 SETM

Maurice Houtsma & Arun Swami:
"Set-oriented mining of association rules"
Research Report RJ 9567
IBM Almaden Research Center, San Jose, California, October 1993

El algoritmo SETM [*SET-oriented Mining of association rules*], al igual que el algoritmo AIS, fue desarrollado en el IBM Almaden Research Center (California). Se deriva del algoritmo AIS y se caracteriza porque fue diseñado pensando utilizar SQL para la generación de itemsets relevantes.

Tal como hacía AIS, SETM genera candidatos en cada iteración conforme va realizando un recorrido secuencial de la base de datos. El algoritmo genera exactamente el mismo conjunto de candidatos que generaba AIS. No obstante, para utilizar SQL estándar, SETM separa la generación de itemsets candidatos de la obtención de la relevancia de los mismos.

SETM utiliza una tabla auxiliar $CT[k]$ en la que almacena los k-itemsets candidatos generados acompañados del identificador de la transacción a partir de la que se obtuvieron (TID). Al final de cada pasada, se obtiene la relevancia de los candidatos ordenando la tabla $CT[k]$ y agrupando las tuplas correspondientes a un mismo itemset.

SETM almacena los identificadores de las transacciones (TID) con los candidatos para evitar tener que comprobar si un itemset está incluido en una transacción dada. De esta forma se acelera el proceso de obtener los itemsets relevantes incluidos en una transacción determinada.

El algoritmo utiliza otra tabla auxiliar, $LT[k]$, para almacenar los k-itemsets relevantes (acompañados de los TIDs correspondientes). Esta tabla se obtiene a partir de $CT[k]$ eliminando de esta tabla aquellos candidatos cuya relevancia no alcanza el umbral preestablecido *MinSupport*.

Asumiendo que la base de datos se encuentra ordenada según TID, SETM puede encontrar los itemsets relevantes incluidos en una transacción ordenando la tabla $LT[k]$ según TID. De esta forma sólo tiene que recorrer una vez la tabla $LT[k]$ para generar los candidatos $CT[k+1]$.

El principal inconveniente de SETM reside en el tamaño que puede llegar a tener la tabla de candidatos $CT[k]$. Para cada itemset candidato (que pueden ser muchos de por sí) en la tabla $CT[k]$ aparecerán tantas tuplas como transacciones haya en la base de datos que incluyan al candidato.

Además, cuando queremos obtener la relevancia de los itemsets candidatos, $CT[k]$ debe reordenarse ya que tras el paso anterior se encontraba ordenada según TID. Para calcular la relevancia de los itemsets la tabla debe ordenarse de forma que todas las tuplas en las que aparezca un itemset estén agrupadas.

Por si todo esto fuera poco, tras eliminar los candidatos cuya relevancia no alcanza el umbral mínimo preestablecido, el conjunto obtenido *LT[k]* necesita reordenarse según TID para poder ser utilizado en la generación de candidatos *CT[k+1]* de la siguiente iteración.

El **algoritmo SETM** es el siguiente:

L[k] y C[k]: Itemsets (igual que en el algoritmo AIS).

LT[k] y CT[k]: Conjuntos de elementos de la forma <TID,itemset> (itemsets junto con los TIDs de las transacciones en las que aparecen)

```
L[1]  = { large 1-itemsets }
LT[1] = { large 1-itemsets acompañados de los TIDs correspondientes }
k = 2
Mientras L[k-1] ≠ ∅
      CT[k] = ∅
      Para cada transacción t ∈ D
            L_t = Conjunto de (k-1)-itemsets de LT[k-1] contenidos en t (TID)
            Para cada itemset l_t ∈ L_t
                  C_t = Extensiones de l_t contenidas en t
                  CT[k] + = { <TID,c> | c ∈ C_t }
      Ordenar CT[k] por itemsets
      Eliminar de CT[k] los itemsets con contador < MinSupport ⇒ LT[k]
      L[k] = { < l.itemset, l.contador > | l ∈ LT[k] }
      Ordenar LT[k] según TID
      k++
Solución:  ∪ L[k]
```

NOTA: SETM se diseñó para utilizar SQL, por lo que las transacciones se representan por conjuntos de tuplas $<TID, item>$.

Importante:

Como se ha comentado, la principal desventaja de SETM reside en el tamaño de los conjuntos *CT[k]* ya que cada conjunto candidato de items aparece tantas veces como el número de transacciones en las que está presente. Si *CT[k]* no cabe en memoria, las dos reordenaciones necesarias en cada iteración deberán realizarse según algún algoritmo de ordenación externa.

EJEMPLO:

Supongamos que partimos de la información obtenida de la base de datos del supermercado. La tabla de datos de entrada al algoritmo SETM será la mostrada a la izquierda (utilizando el nombre del cliente como TID). Como en AIS, obtenemos primero ***L[1]***, formado por los itemsets {Patatas}[2], {Pan}[3], {Huevos}[3] y {Leche}[3]. Además, tenemos que generar la **tabla *LT[1]*** (columna derecha).

Tabla de datos

TID	Ítem
Ana	Leche
Ana	Pan
Juan	Bacon
Juan	Huevos
Juan	Patatas
Luis	Huevos
Luis	Leche
Luis	Pan
Luis	Patatas
María	Huevos
María	Leche
María	Pan

Tabla LT[1]

TID	Itemset
Ana	{Leche}
Ana	{Pan}
Juan	{Huevos}
Juan	{Patatas}
Luis	{Huevos}
Luis	{Leche}
Luis	{Pan}
Luis	{Patatas}
María	{Huevos}
María	{Leche}
María	{Pan}

A partir de la tabla *LT[1]* rellenamos la **tabla *CT[2]***:

TID	Itemset
Ana	{Leche,Pan}
Juan	{Bacon,Huevos}
Juan	{Bacon,Patatas}
Juan	{Huevos,Patatas}
Luis	{Huevos,Leche}
Luis	{Huevos,Pan}
Luis	{Huevos,Patatas}
Luis	{Leche,Pan}
Luis	{Leche,Patatas}
Luis	{Pan,Patatas}
María	{Huevos,Leche}
María	{Huevos,Pan}
María	{Leche,Pan}

Una vez obtenida esta tabla, se ordena por itemsets y se eliminan aquéllos cuya relevancia no llegue a *MinSupport*. Esta operación nos permite obtener la **tabla *LT[2]***:

TID	Itemset
~~Juan~~	~~{Bacon,Huevos}~~
~~Juan~~	~~{Bacon,Patatas}~~
Luis	{Huevos,Leche}
María	{Huevos,Leche}
Luis	{Huevos,Pan}
María	{Huevos,Pan}
Juan	{Huevos,Patatas}
Luis	{Huevos,Patatas}
Ana	{Leche,Pan}
María	{Leche,Pan}
Luis	{Leche,Pan}
~~Luis~~	~~{Leche,Patatas}~~
~~Luis~~	~~{Pan,Patatas}~~

De la tabla LT[2] se obtiene con facilidad L[2]: {Huevos,Leche}[2], {Huevos,Pan}[2], {Huevos,Patatas}[2] y {Leche,Pan}[3].

Para seguir con la siguiente iteración debemos ordenar *LT[2]* según su TID:

TID	Itemset
Ana	{Leche,Pan}
Juan	{Huevos,Patatas}
Luis	{Huevos,Leche}
Luis	{Huevos,Pan}
Luis	{Huevos,Patatas}
Luis	{Leche,Pan}
María	{Huevos,Leche}
María	{Huevos,Pan}
María	{Leche,Pan}

A partir de esta tabla, se genera la **tabla de candidatos *CT[3]***:

TID	Itemset
Juan	{Bacon,Huevos,Patatas}
Luis	{Huevos,Leche,Pan}
Luis	{Huevos,Leche,Patatas}
Luis	{Huevos,Pan,Patatas}
Luis	{Leche,Pan,Patatas}
María	{Huevos,Leche,Pan}

Reordenando y suprimiendo los itemsets que no llegan a la relevancia mínima (*MinSupport*) rellenamos la **tabla *LT[3]***:

TID	Itemset
~~Juan~~	~~{Bacon,Huevos,Patatas}~~
Luis	{Huevos,Leche,Pan}
María	{Huevos,Leche,Pan}
~~Luis~~	~~{Huevos,Leche,Patatas}~~
~~Luis~~	~~{Huevos,Pan,Patatas}~~
~~Luis~~	~~{Leche,Pan,Patatas}~~

De la tabla *LT[3]* se obtiene con facilidad el conjunto *L[3]*, que está formado única y exclusivamente por el itemset {Huevos,Leche,Pan}[2].

Para la siguiente iteración, debemos ordenar nuevamente *LT[3]*, esta vez según TID:

TID	Itemset
Luis	{Huevos,Leche,Pan}
María	{Huevos,Leche,Pan}

A partir de esta tabla se crea la **tabla *CT[4]***:

TID	Itemset
Luis	{Huevos,Leche,Pan,Patatas}

Una vez que tenemos esta tabla, es trivial ver que la **tabla *LT[4]*** quedará vacía, ya que el único elemento de *CT[4]* no alcanza la relevancia mínima:

TID	Itemset
~~Luis~~	~~{Huevos,Leche,Pan,Patatas}~~

Por lo tanto, el conjunto de itemsets relevantes *L[4]* estará vacío y el algoritmo no tendrá que realizar más iteraciones.

Como se puede apreciar, el algoritmo SETM, aunque de una forma diferente, encuentra exactamente los mismos itemsets candidatos que el algoritmo AIS. A continuación se expondrán otros algoritmos que mejoran el rendimiento tanto de SETM como de AIS.

3.3 Apriori, AprioriTID & AprioriHybrid

Rakesh Agrawal & Ramakrishnan Skirant:
"Fast Algorithms for Mining Association Rules"
IBM Research Report RJ9839
IBM Almaden Research Center, San Jose, California (USA), June 1994

Rakesh Agrawal & John C. Shafer
"Parallel Mining of Association Rules"
IEEE Transactions on Knowledge and Data Engineering
December 1996

Agrawal y Skirant, trabajando para el proyecto *Quest* de IBM [*International Business Machines*] en el IBM Almaden Research Center de San José en California, propusieron en 1994 dos algoritmos que superan con creces a los algoritmos anteriormente conocidos, los ya discutidos AIS y SETM.

Los dos algoritmos propuestos se denominan ***Apriori*** y ***AprioriTID*** y son la base de muchos algoritmos posteriores, un punto de referencia para los algoritmos actuales. Difieren notablemente tanto de AIS como de SETM y demuestran ser más eficientes tanto con datos sintéticos como con datos tomados de la realidad.

Aparte de exponer los dos algoritmos mencionados, los autores proponen como se podrían combinar las mejores características de cada uno de ellos en un único algoritmo, denominado ***AprioriHybrid***, que consigue mejores resultados (un tiempo de ejecución proporcional al número de transacciones de la base de datos).

Los algoritmos de la familia *Apriori* realizan múltiples pasadas sobre la base de datos para obtener los conjuntos de itemsets relevantes. En la primera pasada, 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 implica que se generan innecesariamente itemsets candidatos que de por sí nunca pueden llegar a ser relevantes.

Por su parte, tanto en Apriori como en AprioriTID, los candidatos se generan a partir del conjunto de itemsets relevantes encontrados en la iteración anterior, única y exclusivamente. La idea subyacente es que, 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 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]*.

En los algoritmos de la familia *Apriori* se presupone que los items de cada transacción están ordenados lexicográficamente.

El tamaño de un itemset es el número de items que incluye. Un k-itemset es un itemset de tamaño k. Los items dentro de un itemset también se encuentran ordenados ascendentemente. Asociado con cada itemset se haya asociado un contador (para calcular la relevancia del itemset) que inicialmente vale 0.

3.3.1 Algoritmo Apriori

En la primera pasada del algoritmo por la base de datos simplemente se cuenta el número de ocurrencias de los items individuales para determinar el conjunto *L[1]*.

Las iteraciones siguientes se dividen en dos fases. En la iteración *k*, primero se genera el conjunto de candidatos *C[k]* a partir del conjunto *L[k-1]* obtenido en la iteración *k-1*. A continuación, se recorre la base de datos secuencialmente para obtener la relevancia de los elementos del conjunto *C[k]*. Finalmente, incluimos en el conjunto *L[k]* sólo con los itemsets candidatos de *C[k]* que son relevantes (relevancia≥*MinSupport*).

Algoritmo Apriori

```
L[k] es el conjunto de itemsets relevantes que contienen k items.
C[k] es el conjunto de k-itemsets candidatos.
```

```
L[1] = { large 1-itemsets }
k = 2
Mientras L[k-1] ≠ ∅
      C[k] = CandidatosAPRIORI ( L[k-1] )
      Para cada transacción t ∈ D
            Ct = Conjunto de candidatos de C[k] contenidos en t
            Para cada candidato c ∈ Ct
                  c.contador ++
      L[k] = { c ∈ C[k] | c.contador ≥ MinSupport }
      k++

Solución:  ∪ L[k]
```

Generación de candidatos 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. Acto seguido se eliminan del conjunto de candidatos aquellos itemsets que contienen algún (k-1)-itemset que no se encuentre en *L[k-1]*.

```
insert into Ck
select p.item1, p.item2, ..., p.item(k-1), q.item(k-1)
from L[k-1] p, L[k-1] q
where p.item1=q.item1 ... p.item(K-2)=q.item(K-2), p.item(k-1)<q.item(k-1)

Para cada c de C[k], eliminar c de C[k]
si tiene algún subconjunto de k-1 items que no pertenece a L[k-1]
```

vg:

Supongamos *L[3]={{1 2 3},{1 2 4},{1 3 4},{1 3 5},{2 3 4}}*. Tras la reunión, *C[4]* será *{{1 2 3 4},{1 3 4 5}}*. La poda del conjunto de candidatos eliminará el itemset *{1 3 4 5}* porque el itemset *{1 4 5}* no está en *L[3]*. Por lo tanto, *C[4]* sólo incluirá a *{1 2 3 4}*.

Para este ejemplo, si en la base de datos existe una transacción que incluya *{1 2 3 4 5}*, los algoritmos *AIS* y *SETM* generarían los itemsets *{1 2 3 4}* y *{1 2 3 5}* extendiendo *{1 2 3}*. Aparte de ellos, incluirían en *C[4]* tres candidatos más derivados de otros de los elementos de *L[3]* incluidos en la transacción. *AIS* y *SETM* generan un conjunto de candidatos con 5 elementos mientras que *Apriori* sólo incluye un itemset en *C[4]* porque, a priori [de ahí el nombre del algoritmo], se percata de que los otros candidatos nunca pueden llegar a tener la relevancia mínima requerida para ser elementos de *L[4]*.

☞ Independientemente del trabajo de Agrawal y Skirant en IBM, en la Universidad de Helsinki, Mannila, Toivonen y Verkamo propusieron un procedimiento alternativo para la generación de candidatos [véase el apartado correspondiente al algoritmo *OCD*].

Corrección del método empleado:

El conjunto de candidatos *C[k]* generado contiene necesariamente al conjunto de itemsets relevantes *L[k]*, esto es, $C[k] \supseteq L[k]$. Obviamente, cualquier subconjunto de un itemset relevante también tendrá relevancia mínima. Por lo tanto, si extendemos cada itemset de *L[k-1]* con todos los items posibles y eliminamos aquéllos que incluyan algún (k-1)-itemset no incluido en *L[k-1]*, obtendremos un superconjunto de *L[k]*. Esto es lo que hace el procedimiento expuesto. La comprobación `p.item(k-1)<q.item(k-1)` asegura que no se generan itemsets duplicados.

EJEMPLO: Algoritmo Apriori

Supongamos que nuevamente disponemos de la conocida base de datos del supermercado y establecemos el umbral *MinSupport* en 0.5 (para que un ítem sea relevante ha de estar incluido en, al menos, dos de las cuatro transacciones):

Cliente	Artículos
Ana	Leche, Pan
Juan	Bacon, Huevos, Patatas
Luis	Huevos, Leche, Pan, Patatas
María	Huevos, Leche, Pan

Como siempre, inicialmente obtenemos el conjunto ***L[1]*** de items relevantes:

L[1]
{Huevos} [3]
{Leche} [3]
{Pan} [3]
{Patatas} [2]

Primera iteración: Obtención de L[2]

A partir de *L[1]* se genera el conjunto de candidatos ***C[2]***:

C[2]
{Huevos, Leche}
{Huevos, Pan}
{Huevos, Patatas}
{Leche, Pan}
{Leche, Patatas}
{Pan, Patatas}

✔ Nótese que los algoritmos SETM y AIS generaban ocho itemsets candidatos, dos de más.

Se obtiene la relevancia de cada candidato y se genera ***L[2]*** eliminando aquellos itemsets cuya relevancia no alcance el umbral establecido *MinSupport*:

L[2]
{Huevos, Leche} [2]
{Huevos, Pan} [2]
{Huevos, Patatas} [2]
{Leche, Pan} [3]
~~{Leche, Patatas} [1]~~
~~{Pan, Patatas} [1]~~

Segunda iteración: Obtención de L[3]

A partir del conjunto de itemsets relevantes *L[2]* se genera el conjunto de candidatos ***C[3]***. En primer lugar se realiza la operación:

```
insert into C[3] select p.item1, p.item2, q.item2
from L[2] p, L[2] q
where p.item1=q.item1 and p.item2<q.item2
```

Estado inicial de C[3]
{Huevos, Leche, Pan}
{Huevos, Leche, Patatas}
{Huevos, Pan, Patatas}

A continuación se eliminan el segundo itemset porque *{Leche,Patatas}* no pertenece a *L[2]* y el tercero porque *{Pan,Patatas}* tampoco está en *L[2]*. Después obtenemos la relevancia del único itemset que nos queda y comprobamos que alcanza *MinSupport*, por lo que lo incluimos en ***L[3]***:

L[3] = C[3]
{Huevos, Leche, Pan} [2]

Los algoritmos AIS y SETM habrían generado cinco itemsets candidatos, mientras que *Apriori* sólo genera uno (que de hecho es relevante).

Tal como aparece descrito el algoritmo, aún se intentaría generar *C[4]*, que, como es evidente, no contendrá ningún elemento.

Ahora se describirán brevemente tres alternativas propuestas por Agrawal y Shafer (*"Parallel Mining of Association Rules"*, IEEE Transactions on Knowledge and Data Engineering, December 1996) para implementar el algoritmo *Apriori* en un multiprocesador sin memoria compartida. Las técnicas expuestas son, por lo tanto, aplicables a un cluster de estaciones de trabajo.

COUNT DISTRIBUTION

Este algoritmo realiza cálculos redundantes para reducir la comunicación necesaria entre los distintos procesadores. Cada uno de los procesadores dispone de una parte de la base de datos global en su disco local.

Inicialmente, cada procesador P_i genera un conjunto de candidatos $C_i[1]$ a partir de los datos almacenados en su disco local. La información obtenida se distribuye y se obtiene $L[1]$. En cada iteración posterior, cada procesador genera el conjunto completo de candidatos $C[k]$ a partir del conjunto $L[k-1]$, del que todos tienen una copia. Cada procesador recorre su parte de la base de datos y obtiene la relevancia local de cada itemset candidato de $C[k]$. Las relevancias locales se intercambian entre los procesadores para obtener la relevancia real de cada candidato. Finalmente, cada procesador obtiene $L[k]$ a partir de $C[k]$.

DATA DISTRIBUTION

Este algoritmo fue ideado para aprovechar la capacidad total de memoria disponible en el sistema multiprocesador (*Count Distribution* no supone ninguna mejora en ese sentido respecto al algoritmo serie). En cambio, requiere mayor velocidad de comunicación entre los distintos procesadores: cada procesador debe difundir su información local a todos los demás procesadores en cada iteración.

El procesador *i* genera el conjunto $C[k]$ a partir de $L[k-1]$ pero esta vez almacena únicamente una parte $C_i[k]$ del conjunto (se puede usar una estrategia *round-robin* para asignar itemsets equitativamente a cada procesador sin necesidad de comunicación). El procesador *i* obtiene la relevancia de cada itemset de $C_i[k]$ con sus transacciones y las que le envían los demás procesadores. A continuación, cada procesador genera un conjunto local de itemsets relevantes $L_i[k]$ a partir de $C_i[k]$ y la información obtenida se distribuye de forma que todos los procesadores dispongan de $L[k]$ completo para la siguiente iteración.

CANDIDATE DISTRIBUTION

Este algoritmo intenta explotar información propia del dominio del problema distribuyendo tanto las transacciones como los itemsets entre los distintos procesadores.

Los itemsets relevantes de $L[k-1]$ se distribuyen de forma que cada procesador genere un subconjunto de $C[k]$ tal que $C_i[k] \cap C_j[k] = \varnothing$ si $i \neq j$. Además, cada procesador i dispone en su disco local de todas las transacciones que necesita para obtener la relevancia de los itemsets $C_i[k]$. Nótese que esto implica que parte de la base de datos estará repetida en varios procesadores. Se reduce de esta forma la comunicación entre procesadores realizada en *Data Distribution*. Además, cada procesador dispone de información acerca de los procesadores interesados en la relevancia de cada itemset para realizar la poda de $C[k]$.

OTRAS ALTERNATIVAS

Además del estudio realizado por Agrawal y Shafer, otros investigadores también han tratado el problema de la obtención de reglas de asociación en bases de datos distribuidas (totalmente equivalente a la paralelización de los algoritmos secuenciales de obtención de reglas de asociación).

Por ejemplo, Cheung, Han, Ng, Fu y Fu proponen un algoritmo denominado **FDM** [*Fast Distributed Mining of Association Rules*] que reduce la comunicación necesaria en *Count Distribution* [*CD*] aprovechando relaciones existentes entre los itemsets relevantes localmente (en la base de datos local) y los itemsets relevantes globalmente (en la base de datos completa). En su artículo *"A Fast Distributed Algorithm for Mining Association Rules"*, de PDIS'96, exponen distintas variantes del algoritmo: *FDM-LP* [*FDM with Local Prunning*], *FDM-LUP* [*FDM with Local and Upper-Bound Prunning*] y *FDM-LPP* [*FDM with Local Prunning and Polling-Site Prunning*].

3.3.2 Algoritmo AprioriTID

El algoritmo *AprioriTID* se caracteriza porque no accede a la base de datos para obtener la relevancia de los candidatos. Para ello utiliza los conjuntos auxiliares *CT[k]*.

Cada miembro del conjunto auxiliar *CT[k]* es de la forma $<TID, \{X\}>$, donde cada *X* es un k-itemset potencialmente relevante (un candidato) presente en la transacción identificada por *TID*. Evidentemente, *CT[1]* se corresponde a la base de datos original en la cual cada item *i* es reemplazado por el itemset *{i}*. El elemento de *CT[k]* correspondiente a la transacción *t* es el par $< TID_t, \{ c \in C[k] \mid c \subseteq t \} >$. Si una transacción no contiene ningún k-itemset candidato no tendrá una entrada en *CT[k]*.

Algoritmo AprioriTID

```
L[k]  es el conjunto de itemsets relevantes que contienen k items.
C[k]  es el conjunto de k-itemsets candidatos.
CT[k] es el conjunto de k-candidatos con sus TIDs asociados.
```

```
L[1] = { large 1-itemsets }

CT[1] = Base de datos D

k = 2

Mientras L[k-1] ≠ ∅

      C[k] = candidatosAPRIORI ( L[k-1] )

      CT[k] = ∅

      Para cada entrada t ∈ CT[k-1]

            Ct = Conjunto de candidatos de C[k] contenidos en t (usando TID)

            Para cada candidato c ∈ Ct

                  c.contador ++

            Si Ct ≠ ∅

                  CT[k] += <t.TID, Ct>

      L[k] = { c ∈ C[k] | c.contador ≥ MinSupport }

      k++

Solución:  ∪ L[k]
```

La característica principal de *AprioriTID* es que, en cada iteración, se recorre el conjunto *CT[k-1]* en vez de la base de datos completa para obtener la relevancia de los itemsets de *C[k]*. En la generación de candidatos, el algoritmo *AprioriTID* utiliza el método ya comentado para el algoritmo *Apriori*.

El tamaño de los conjuntos auxiliares *CT[k]* puede llegar a ser mucho menor que el de la base de datos original tras unas cuantas iteraciones del algoritmo (para valores grandes de k), lo que ahorra esfuerzo consumido realizando operaciones de E/S.

Sin embargo, para valores pequeños de k (especialmente cuando k vale 2 ó 3), las entradas correspondientes a cada transacción en *CT[k]* pueden ocupar más espacio que las propias transacciones en la base de datos original: los conjuntos *CT[k]* pueden llegar a ser mayores que la base de datos inicial para valores pequeños de k.

Corrección del algoritmo:

Diremos que *CT[k]* es completo si $\forall t \in CT[k]$ el conjunto de itemsets *t.itemsets* incluye todos los k-itemsets incluidos en la transacción *t.TID*. *CT[k]* es correcto si $\forall t \in CT[k]$ el conjunto de itemsets *t.itemsets* no incluye ningún k-itemset que no esté incluido en la transacción cuyo identificador es *t.TID*.

LEMA: $\forall k>1$, si *CT[k-1]* es completo además de correcto y *L[k-1]* es el correcto, entonces el conjunto C_t generado por *AprioriTID* en la iteración *k* es el conjunto de k-itemsets candidatos de *C[k]* incluidos en la transacción *t.TID*.

LEMA: $\forall k>1$, si *L[k-1]* es el correcto y el conjunto C_t generado por *AprioriTID* en la iteración *k* es el conjunto de k-itemsets candidatos de *C[k]* incluidos en la transacción identificada por *t.TID*, entonces el conjunto *CT[k]* es correcto y completo.

TEOREMA: $\forall k>1$, el conjunto C_t generado por *AprioriTID* en la iteración *k* es el conjunto de k-itemsets candidatos de *C[k]* incluidos en la transacción identificada por *t.TID*.

Primero se demuestra por inducción que *L[k]* es correcto y *CT[k]* es correcto y completo para todo $k \geq 1$. Cuando $k=1$, la demostración es trivial: *CT[1]* se corresponde con la base de datos inicial y *L[1]* es correcto por definición. Del primer lema obtenemos que el conjunto C_t generado en la iteración $n+1$ es el conjunto de (n+1)-itemsets candidatos de *C[n+1]* incluidos en la transacción *t.TID*. Ya que la función de generación de candidatos garantiza que $C[n+1] \supseteq L[n+1]$ y C_t es el correcto, entonces *L[n+1]* será el correcto. Del segundo lema concluimos que *CT[n+1]* será correcto y completo.

Al ser CT[k] correcto y completo además de ser L[k] el correcto para todo $k \geq 1$, el teorema se demuestra haciendo uso del primer lema.

EJEMPLO: Algoritmo AprioriTID

Partamos una vez más del ejemplo de la base de datos del supermercado y establezcamos *MinSupport* en 0.5:

TID	Items
Ana	Leche, Pan
Juan	Bacon, Huevos, Patatas
Luis	Huevos, Leche, Pan, Patatas
María	Huevos, Leche, Pan

Primero tendremos que generar el conjunto ***CT[1]***:

TID	Itemsets
Ana	{ {Leche}, {Pan} }
Juan	{ {Bacon}, {Huevos}, {Patatas} }
Luis	{ {Huevos}, {Leche}, {Pan}, {Patatas} }
María	{ {Huevos}, {Leche}, {Pan} }

Obtenemos el conjunto de itemsets relevantes ***L[1]***:

Itemset	Support
{Huevos}	3
{Leche}	3
{Pan}	3
{Patatas}	2

Generamos los itemsets candidatos de ***C[2]*** como en *Apriori*:

Itemset
{Huevos, Leche}
{Huevos, Pan}
{Huevos, Patatas}
{Leche, Pan}
{Leche, Patatas}
{Pan, Patatas}

En ***CT[2]*** tenemos:

TID	Itemsets
Ana	{ {Leche,Pan} }
Juan	{ {Huevos,Patatas} }
Luis	{ {Huevos,Leche}, {Huevos,Pan}, {Huevos,Patatas}, {Leche,Pan}, {Leche,Patatas}, {Pan,Patatas} }
María	{ {Huevos,Leche}, {Huevos,Pan}, {Leche,Pan} }

Creamos ***L[2]***:

Itemset	*Support*
{Huevos, Leche}	2
{Huevos, Pan}	2
{Huevos, Patatas}	2
{Leche, Pan}	3

Generamos ***C[3]*** a partir de *L[2]*:

Itemset
{Huevos, Leche,Pan}

Obtenemos ***CT[3]***:

TID	Itemsets
Luis	{ {Huevos,Leche,Pan} }
María	{ {Huevos,Leche,Pan} }

Finalmente conseguimos ***L[3]***:

Itemset	*Support*
{Huevos, Leche,Pan}	2

Cuando se genera ***C[4]*** a partir de *L[3]* obtenemos un conjunto vacío, con lo que el algoritmo termina.

3.3.3 Algoritmo AprioriHybrid

"BEAUTY: The adjustment of all parts proportionately so that one cannot add or subtract or change without impairing the harmony of the whole"

"BELLEZA: Ajuste proporcional de todas las partes de tal modo que no se puede añadir, eliminar o cambiar algo sin estropear la armonía del conjunto"

Leon Battista Alberti

El algoritmo ***AprioriHybrid*** consiste en combinar los algoritmos *Apriori* y *AprioriTID* de una forma adecuada, conservando lo mejor de cada uno de ellos.

En las primeras iteraciones *Apriori* se comporta mejor que *AprioriTID*, ya que los conjuntos *CT[k]* utilizados por este último pueden ser mayores que la base de datos inicial. No obstante, cuando *k* aumenta el número de itemsets candidatos se reduce y, mientras que *Apriori* sigue recorriendo la base de datos completa, *AprioriTID* recorre simplemente los conjuntos *CT[k]*. El tamaño de los conjuntos *CT[k]* recorridos por *AprioriTID* es mucho menor que el de la base de datos completa cuando *k* es elevado.

Como norma general, se puede utilizar *AprioriTID* en cuanto el conjunto auxiliar *CT[k]* quepa en memoria. Cuando *CT[k]* cabe en memoria principal es más eficiente utilizar *AprioriTID*, ya que no tenemos que realizar operaciones de E/S sobre disco.

El algoritmo *AprioriHybrid* suele comportarse mejor que *Apriori* en la mayor parte de los casos, aunque la mejora puede no ser significativa (el verdadero cuello de botella de estos algoritmos se encuentra en las primeras iteraciones).

3.4 OCD [Offline Candidate Determination]

Heikki Mannila, Hannu Toivonen & A. Inkeri Verkamo
"Improved Methods for Finding Association Rules"
Department of Computer Science, University of Helsinki, C-1993-65
Helsinki, December 1993

H. Mannila, H. Toivonen & A. I. Verkamo
"Efficient Algorithms for Discovering Association Rules"
AAAI'94 Workshop on Knowledge Discovery in Databases (KDD'94)
Seattle, Washington, USA, July 1994.

En la Universidad de Helsinki, Mannila, Toivonen y Verkamo idearon un algoritmo similar al algoritmo *Apriori*. Este algoritmo fue desarrollado independientemente del trabajo de Agrawal y Skirant en IBM y se caracteriza porque también aprovecha el hecho de que cualquier subconjunto de un itemset relevante es a su vez un itemset relevante.

Como se verá a continuación, el algoritmo *OCD* supone una mejora respecto al algoritmo *AIS* pero no es superior al algoritmo *Apriori*. *OCD* difiere del ya expuesto *Apriori* en dos aspectos: la forma de obtener los conjuntos de candidatos *C[k]* y la estructura de datos empleada (*OCD* no utiliza como estructura de datos un árbol hash).

Generación de candidatos en OCD

El conjunto de candidatos *C[k]* es el conjunto *{ X⊆U | #X=k y X incluye k miembros de L[k-1] }*. Esta no es más que una forma alternativa de expresar que cualquier subconjunto de un itemset relevante es también un itemset relevante. De hecho, un elemento del conjunto *L[k+e]* incluye $\binom{k+e}{k}$ k-itemsets relevantes de *L[k]*. Como caso particular, un itemset relevante de *L[k+1]* debe incluir exactamente (k+1) itemsets de *L[k]*: $\binom{k+1}{k} = k+1$

Utilizando la misma propiedad podemos llegar a otro resultado interesante. Podemos afirmar que el conjunto *L[k+e]* de itemsets relevantes estará vacío si *L[k]* contiene menos de $\binom{k+}{k}$ k-itemsets relevantes.

Los autores del algoritmo *OCD* propusieron dos fórmulas alternativas de obtener el conjunto de candidatos *C[k]*, siendo la segunda opción más restrictiva. En la primera de las alternativas se obtienen los candidatos potenciales *C'[k]* a partir de elementos de *L[k-1]* y *L[1]*, mientras que en la segunda se obtienen a partir de pares de itemsets de *L[k-1]* que tengan *(k-2)* items en común.

✗ *Primera alternativa*

```
C'[k] = { X∪Y | X ∈ L[k-1] & Y ∈ L[1] & Y ∉ X }
C[k]  = { X | X en C'[k] & X contiene k miembros de L[k-1] }
```

✓ *Segunda alternativa*

```
C'[k] = { X∪Y | X,Y ∈ L[k-1] & #(X∩Y) = k-2 }
C[k]  = { X | X en C'[k] & X contiene k miembros de L[k-1] }
```

Las dos alternativas propuestas tienen en común la segunda parte, la generación del conjunto final de candidatos *C[k]* eliminando aquéllos k-itemsets que no incluyen k miembros del conjunto de itemsets relevantes *L[k-1]*. Esta restricción es equivalente a la poda del conjunto C[k] que se realiza en el algoritmo *Apriori*: se eliminan de *C[k]* a priori aquellos k-itemsets que incluyen algún (k-1)-itemset no incluido en *L[k-1]* ya que el k-itemset nunca puede llegar a ser un itemset relevante.

Para el paso inicial de la generación de candidatos, Mannila, Toivonen y Verkamo proponen dos formas de obtener el conjunto *C'[k]*, siendo la segunda opción más restrictiva. El conjunto generado por la primera alternativa es un superconjunto del generado por la segunda. A su vez, el conjunto *C'[k]* generado por la segunda alternativa propuesta es un superconjunto del conjunto creado por *Apriori* en la primera fase de la generación de *C[k]* (antes de la eliminación de aquellos posibles "candidatos" que incluyen algún *(k-1)*-itemset no perteneciente a *L[k-1]*). Por lo tanto, el algoritmo *OCD* no es mejor que *Apriori*.

vg:

Retomemos el ejemplo propuesto al exponer el algoritmo *Apriori*. Partiendo del conjunto de itemsets relevantes $L[3]=\{\{1\ 2\ 3\},\{1\ 2\ 4\},\{1\ 3\ 4\},\{1\ 3\ 5\},\{2\ 3\ 4\}\}$ se obtiene, aplicando el método utilizado por el algoritmo *Apriori* o cualquiera de las alternativas propuestas para *OCD*, el conjunto de candidatos $C[4] = \{\{1\ 2\ 3\ 4\}\}$.

La diferencia estriba en el conjunto *C'[4]*. Si utilizamos la primera opción propuesta para *OCD*, *C'[4]* incluirá *{1 2 3 4}*, *{1 2 3 5}*, *{1 2 4 5}*, *{1 3 4 5}* y *{2 3 4 5}*. Si utilizamos la segunda técnica descrita deberíamos incluir en el conjunto auxiliar *C'[4]* los itemsets *{1 2 3 4}*, *{1 2 3 5}* y *{1 3 4 5}*, los cuales se pueden derivar de las uniones de itemsets relevantes $\{1\ 2\ 3\}\cup\{1\ 2\ 4\}$, $\{1\ 2\ 3\}\cup\{1\ 3\ 5\}$ y $\{1\ 3\ 4\}\cup\{1\ 3\ 5\}$, respectivamente. En *Apriori*, el conjunto equivalente a *C'[4]* contendría únicamente a los itemsets *{1 2 3 4}* y *{1 3 4 5}*.

3.5 DHP [Direct Hashing and Prunning]

Jong Soo Park, Ming-Syan Chen & Philip S. Yu
"An Effective Hash-Based Algorithm for Mining Association Rules"
Proceedings of the ACM SIGMOD Conference on Management of Data
San Jose, California, USA, May 1995

Jong Soo Park, Ming-Syan Chen & Philip S. Yu
"Using a Hash-Based Method with Transaction Trimming for Mining Association Rules"
IEEE Transactions on Knowledge and Data Engineering
September/October 1997.

Cuantos más itemsets se incluyan en el conjunto de candidatos más tardará cualquier algoritmo en encontrar los itemsets relevantes. La idea básica introducida por el algoritmo *DHP* es la utilización de una heurística que genere únicamente aquellos itemsets candidatos con una probabilidad elevada de ser relevantes.

El algoritmo *DHP*, construido a partir del algoritmo *Apriori*, procura reducir el número de candidatos generados utilizando una tabla hash para (k+1)-itemsets al generar *L[k]*. Esta tabla hash debe utilizarse sobre todo en las primeras iteraciones, que son las que producen mayor número de candidatos y, por ende, mayor coste computacional a la hora de obtener los conjuntos *L[k]* correspondientes.

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* derivados del itemset *i* tal como se describe en el algoritmo *Apriori*.

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]*.

EJEMPLO:

Utilicemos una vez más el ejemplo de la base de datos que incluye información acerca de las compras realizadas en un supermercado:

Cliente	Artículos
Ana	Leche, Pan
Juan	Bacon, Huevos, Patatas
Luis	Huevos, Leche, Pan, Patatas
María	Huevos, Leche, Pan

Como siempre, inicialmente obtenemos el conjunto ***L[1]*** de items relevantes:

L[1]
{Huevos} [3]
{Leche} [3]
{Pan} [3]
{Patatas} [2]

Supongamos que disponemos de la siguiente función hash:

Item	*h*
Bacon	0
Huevos	1
Leche	2
Pan	3
Patatas	4

Podemos definir recursivamente la función hash para un itemset:

$$h(\{item\} \cup ITEMSET) = (h(item)*5 + h(ITEMSET)) \bmod 7$$

donde 5 es el número de items diferentes y 7 es el tamaño de la tabla hash utilizada.

De cada una de las transacciones de la base de datos vamos obteniendo los itemsets candidatos incluidos en la transacción actual (algo similar a la obtención del conjunto *CT[2]* en el algoritmo *AprioriTID*) e incrementamos los contadores asociados a las entradas de la tabla hash correspondientes a los candidatos. Estos son los candidatos de cada una de las transacciones:

TID	Itemsets candidatos
Ana	{ {Leche,Pan} }
Juan	{ {Bacon,Huevos}, {Bacon,Patatas}, {Huevos,Patatas} }
Luis	{ {Huevos,Leche}, {Huevos,Pan}, {Huevos,Patatas}, {Leche,Pan}, {Leche,Patatas}, {Pan,Patatas} }
María	{ {Huevos,Leche}, {Huevos,Pan}, {Leche,Pan} }

A partir de ellos obtenemos lo siguiente:

0	1	2	3	4	5	6	Entrada
3	3	2	0	1	1	3	Contador
{Huevos, Leche} {Leche, Patatas} {Huevos, Leche}	{Bacon, Huevos} {Huevos, Pan} {Huevos, Pan}	{Huevos, Patatas} {Huevos, Patatas}		{Bacon, Patatas}	{Pan, Patatas}	{Leche, Pan} {Leche, Pan} {Leche, Pan}	Candidatos

Una vez que tenemos *L[1]*, al generar un candidato miramos en la tabla hash generada anteriormente para decidir si debemos incluirlo en el conjunto de candidatos *C[2]* o no. Aplicando el algoritmo DHP obtendríamos el siguiente conjunto de candidatos ***C[2]***:

C[2]
{Huevos, Leche}
{Huevos, Pan}
{Huevos, Patatas}
{Leche, Pan}
{Leche, Patatas}

La casilla de la tabla hash correspondiente al itemset *{Pan,Patatas}* tiene sólo 1 en su contador, por lo que ya sabemos que el itemset no puede ser relevante y, por lo tanto, no lo incluimos en *C[2]*. Este itemset sí era incluido en *C[2]* por el algoritmo *Apriori* aunque luego, obviamente, era descartado al construir *L[2]*.

En este ejemplo se puede ver como el conjunto de candidatos generado por *DHP* es más reducido que el que generaba el algoritmo *Apriori*.

Obviamente, para que el conjunto de candidatos generado sea óptimo (igual al conjunto de itemsets relevantes), se necesita una función hash perfecta, para lo cual la tabla hash habría de tener N^2 entradas, siendo N el número de items diferentes. Esto equivale a un array bidimensional para contar 2-itemsets a la vez que se obtiene el conjunto *L[1]*, la técnica utilizada por Agrawal y Skirant para obtener eficientemente *L[2]*. No obstante, esta técnica es inviable en cuanto el número de items diferentes es elevado.

DHP es especialmente eficiente en la generación del conjunto de candidatos *C[2]*, donde el algoritmo *AprioriTID* era altamente ineficiente. En el algoritmo *Apriori*, el conjunto *C[2]* incluye $\binom{\#L[1]}{2}$ itemsets de dos elementos. Cuando *#L[1]* es elevado, *C[2]* es un conjunto enorme de itemsets y conviene reducir su tamaño al máximo.

Al igual que *Apriori* se combinaba con *AprioriTID* en el algoritmo *AprioriHybrid*, *DHP* también podría combinarse con *AprioriTID*. *AprioriTID* se vería beneficiado por la mejora sustancial que supone *DHP* respecto a *Apriori* en las primeras pasadas por la base de datos. Por su parte, *DHP* puede beneficiarse de la eficiencia de *AprioriTID* en las siguientes iteraciones.

✳

Otro aspecto importante que afecta al rendimiento de los algoritmos de obtención de itemsets relevantes, como se vio con *AprioriTID*, es el tamaño de la base de datos que ha de recorrerse en cada iteración. La solución directa implica recorrer la base de datos completa en cada iteración pero, conforme *k* aumenta, no sólo disminuye el número de k-itemsets relevantes sino también la cantidad de transacciones que incluyen algún k-itemset relevante.

Al ejecutarse, *DHP* va reduciendo el tamaño de la base de datos en cada iteración. Como ya se ha visto, para que una transacción contenga un (k+1)-itemset relevante, ésta deberá contener *(k+1)* k-itemsets relevantes como mínimo. Por lo tanto, se pueden descartar para las siguientes pasadas aquellas transacciones que contengan menos de *(k+1)* k-itemsets relevantes.

Como se ha visto, DHP tiene dos cualidades interesantes: una generación eficiente de itemsets relevantes reduciendo el número de candidatos y la reducción del tamaño efectivo de la base de datos de transacciones que ha de recorrerse en cada iteración. Esto último tiene su lado negativo porque hay que duplicar la base de datos y se supone que estamos trabajando con bases de datos enormes.

Aparte de lo ya comentado, en su artículo de *IEEE Transactions on Knowledge and Data Engineering*, Park, Chen y Yu proponen que, cuando el número de candidatos $\#C[k]$ no sea excesivamente elevado, $C[k+1]$ se obtenga de $C[k]*C[k]$ directamente, en vez de utilizar $L[k]*L[k]$. Así se puede suprimir un recorrido por la base de datos: en un único recorrido secuencial obtendremos $L[k]$ y $L[k+1]$. En el mejor de los casos, con sólo dos pasadas por la base de datos seremos capaces de obtener los conjuntos de itemsets relevantes (una pasada inicial es necesaria para obtener $L[1]$).

De hecho, Agrawal y Skirant ya propusieron esta técnica en su informe acerca de los algoritmos *Apriori*, *AprioriTID* y *AprioriHybrid*. Se puede comprobar con facilidad que el conjunto de candidatos $C'[k+1]$ generado a partir de $C[k]$ incluye necesariamente a todos los elementos del conjunto $C[k+1]$, que se deriva de $L[k]$: $C'[k+1] \supseteq C[k+1]$. Por lo tanto, la técnica es perfectamente válida y su aplicación dependerá en gran medida de la memoria disponible para almacenar los conjuntos de itemsets candidatos.

3.6 Max-Miner

Roberto J. Bayardo Jr.
"Efficiently Mining Long Patterns from Databases"
IBM Almaden Research Center
ACM PODS [Principles of Database Systems], Seattle, WA, June 1998

Para concluir la exposición de algoritmos para la obtención de itemsets relevantes comentaremos un algoritmo reciente que aborda el problema desde un punto de vista diferente. Los algoritmos analizados hasta ahora resultan adecuados cuando el tamaño de los itemsets relevantes es relativamente pequeño pero no lo son cuando en el dominio del problemas aparecen patrones de longitud elevada (como en el análisis de proteínas o del ADN humano).

Al obtener los itemsets de *L[k]* una vez que ya se han obtenido los de *L[k-1]*, los algoritmos como Apriori necesitan descubrir del orden de 2^k itemsets relevantes antes de encontrar un k-itemset relevante. Cuando k es elevado, esta complejidad exponencial restringe la aplicabilidad del algoritmo.

Denominaremos itemset relevante maximal a aquel itemset relevante que no sea un subconjunto propio de otro itemset relevante. *Max-Miner* obtiene eficientemente el conjunto de itemsets relevantes maximales utilizando heurísticas en su búsqueda.

Max-Miner utiliza un árbol [*set-enumeration tree*] para representar los items (*Rymon: "Search through systematic set enumeration", Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning, 1992*).

Al construir el árbol se supone que existe un orden prefijado entre items, de forma similar a como se ordenaban los items dentro de un itemset en los algoritmos de la familia *Apriori*.

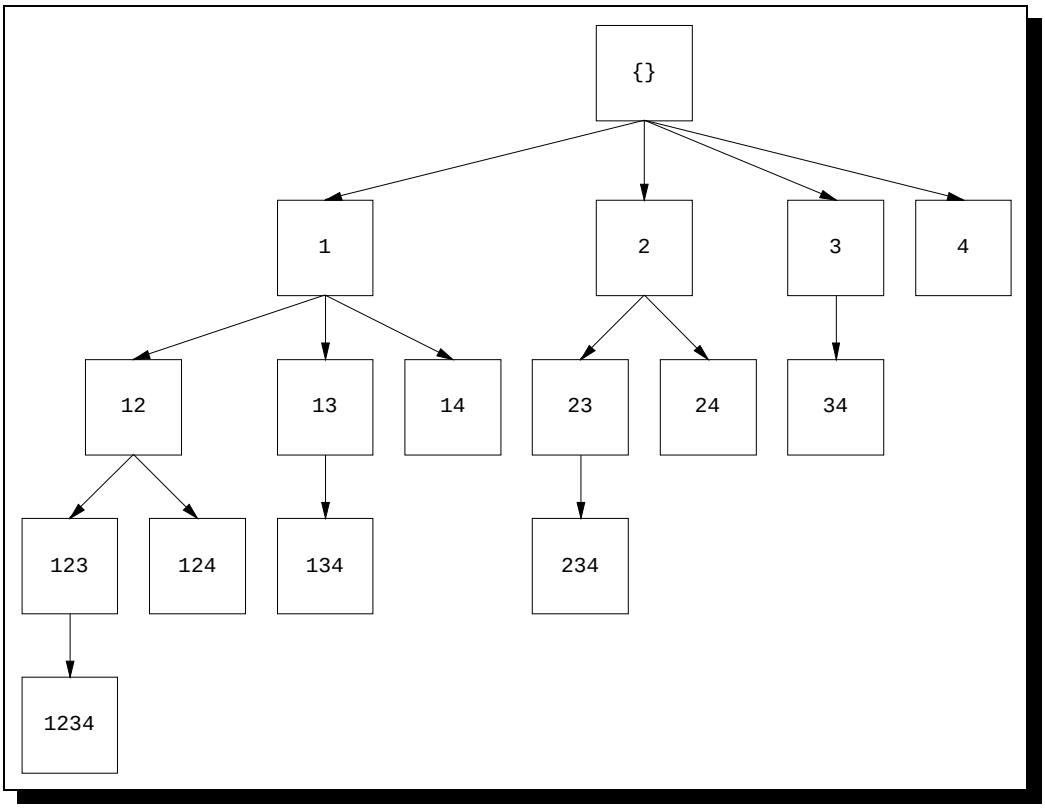
La raíz del árbol corresponde al conjunto vacío. Un nodo situado a una profundidad k está asociado a un conjunto de k elementos (un k-itemset). Los hijos de un nodo situado a profundidad *k-1* siempre son k-itemsets derivados del nodo padre tal como se obtenían candidatos a partir de itemsets relevantes en el algoritmo *Apriori*.

En *Max-Miner*, a cada nodo del árbol de enumeración se le asocia un grupo candidato *g*. Este grupo candidato *g* está formado por dos conjuntos de items:

➢ $h(g)$ [*head*]: El itemset correspondiente al nodo

➢ $t(g)$ [*tail*]: El conjunto de los items que pueden aparecer en los subnodos.

Calcular la relevancia de un grupo candidato *g* es obtener la relevancia de los itemsets $h(g)$, $h(g) \cup t(g)$ y $h(g) \cup \{i\} \ \forall i \in t(g)$.



Árbol de enumeración de conjuntos completo para 4 items diferentes

vg: El grupo candidato *g* asociado al nodo *{1,2}* está formado por $h(g)=\{1,2\}$ y $t(g)=\{3,4\}$.

FUNCIONAMIENTO DEL ALGORITMO:

Inicialmente, se analiza el dominio de los items y se crea el árbol de enumeración (paso equivalente a la obtención de *L[1]* en *Apriori*). A continuación, Max-Miner realiza un recorrido en anchura por el árbol de enumeración para obtener el conjunto de itemsets relevantes maximales. Conforme lo va recorriedo, *Max-Miner* poda el árbol haciendo uso de dos criterios: *superset-frequency prunning* y *subset-infrequency prunning*. Se expanden sólo aquéllos nodos tales que $h(g) \cup t(g)$ no es relevante y no está incluido en ningún itemset de los que son relevantes, tras haber eliminado de $t(g)$ aquellos items que hacen que $h(g) \cup \{i\}$ no sea relevante.

Para aumentar la eficacia de la poda del árbol, los items se colocan en el árbol en orden ascendente de relevancia [*support*]. De esta forma, los items más frecuentes aparecerán en mayor cantidad de grupos candidatos y la poda siguiendo el primer criterio será más efectiva. Por el mismo motivo, en un nodo dado los itemsets $h(g) \cup \{i\}$ se colocan también en orden creciente de relevancia $support(h(g) \cup \{i\})$.

Superset-frequency prunning:

Si $h(g) \cup t(g)$ es relevante, cualquier itemset enumerado por el nodo será relevante pero no maximal (no hace falta seguir expandiendo el nodo).

Subset-infrequency prunning:

Si un itemset $h(g) \cup \{i\}$ no es relevante, entonces cualquier cabeza de un nodo que incluya i tampoco lo será, por lo que se puede suprimir del conjunto $t(g)$.

El tiempo empleado por el algoritmo *Max-Miner* crece linealmente con el número de itemsets relevantes maximales, independientemente de su longitud, mientras que el tiempo empleado por algoritmos derivados de *Apriori* crece exponencialmente con la longitud máxima de los itemsets relevantes.

4. Algoritmos para la obtención de reglas de asociación (Obtención de reglas de asociación a partir de los itemsets relevantes)

Las reglas de asociación se obtienen fácilmente a partir de los conjuntos de itemsets relevantes. Una regla de asociación es una implicación de la forma X⇒Y donde X e Y son conjuntos de items de intersección vacía. Se permitirán reglas con varios items en el consecuente. La fiabilidad [*confidence*] de la regla de asociación X⇒Y es la proporción de transacciones con X que contienen también a Y: $support(X \cup Y)/support(X)$. La relevancia [*support*] de la regla es la proporción de transacciones de la base de datos que contienen tanto a X como a Y, es decir, $support(X \cup Y)$.

Para cada itemset relevante *i* obtenemos todos sus subconjuntos *s*. De cada subconjunto *s* se obtiene una regla de la forma $s \Rightarrow (i-s)$, siendo la fiabilidad de la regla 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.

De un k-itemset relevante podemos obtener hasta 2^k-2 reglas ($\sum_{i=1}^{k-1}\binom{k}{i}$), cantidad bastante considerable si k es elevado. La obtención de reglas interesantes a partir de la gran cantidad de reglas de asociación que se pueden llegar a generar puede considerarse un problema de *Data Mining* (*"un problema de Data Mining de segundo orden"*).

NOTA: Acerca del algoritmo Max-Miner

Para obtener todas las **reglas de asociación** Max-Miner debería realizar una pasada adicional por la base de datos para obtener la relevancia de todos los itemsets relevantes (ya que sólo se han obtenido los maximales). No obstante, *Max-Miner* puede utilizarse para identificar muchas, aunque no todas, las reglas de asociación que tengan una fiabilidad elevada. Tras obtener la relevancia de un grupo candidato *g*, se pueden obtener todas las reglas de la forma $h(g) \Rightarrow i$ siendo *i* un ítem de $t(g)$.

4.1 Algoritmo directo

Para generar las reglas de asociación derivadas de un itemset relevante *l* se ha de consultar la relevancia de todos los subconjuntos del itemset *l*. Para cada uno de esos subconjuntos *a*, se genera una regla si el cociente *support(l)/support(a)* supera el umbral mínimo preestablecido *MinConfidence*.

Para mejorar la eficiencia del algoritmo, se pueden generar los subconjuntos de un itemset de forma recursiva. Si un subconjunto *a* de un itemset *l* no genera una regla $a \Rightarrow (l-a)$ entonces ningún subconjunto propio *s* del itemset *a* generará reglas con la fiabilidad mínima requerida, ya que $support(s) > support(a)$ y, por lo tanto, $support(l)/support(s) \leq support(l)/support(a)$; es decir, la fiabilidad de la regla $a \Rightarrow (l-a)$, que no llegaba a *MinConfidence*, será superior a la de la regla $s \Rightarrow (l-s)$ derivada de *s*.

Algoritmo directo

```
Para cada itemset relevante l_k (k≥2)
      GenerarReglas (l_k, l_k)
```

procedimiento GenerarReglas (l_k , a_m)

```
      A = { (m-1)-itemsets incluidos en a_m }
      Para cada a_m-1 ∈ A
            confidence = support(l_k) / support (a_m-1)
            Si confidence > MinConfidence
                  Regla a_m-1 ⇒ (l_k - a_m-1) [confidence, support(l_k)]
                  Si m>2
                        GenerarReglas (l_k, a_m-1)
```

EJEMPLO: Utilicemos una vez más la base de datos del supermercado como muestra del funcionamiento del algoritmo expuesto, fijando tanto *MinSupport* como *MinConfidence* en *0.5*.

TID	Items
Ana	Leche, Pan
Juan	Bacon, Huevos, Patatas
Luis	Huevos, Leche, Pan, Patatas
María	Huevos, Leche, Pan

A partir de las transacciones realizadas obtenemos los itemsets relevantes siguiendo cualquiera de los algoritmos expuestos (desde AIS hasta DHP):

Itemset	Support
{Huevos}	3
{Leche}	3
{Pan}	3
{Patatas}	2
{Huevos, Leche}	2
{Huevos, Pan}	2
{Huevos, Patatas}	2
{Leche, Pan}	3
{Huevos, Leche, Pan}	2

Para obtener las catorce reglas de asociación derivadas de los itemsets relevantes, el algoritmo hemos de aplicarlo para cada k-itemset relevante (con $k \geq 2$):

❶ l_k = *{Huevos, Leche}*

GenerarReglas ({Huevos, Leche}, {Huevos, Leche})

A = { {Huevos}, {Leche} }

☞ a = { Huevos }
Confidence = Support({Huevos, Leche}) / Support({Huevos}) = 2/3
Regla **{Huevos} ⇒ {Leche} [2/3]**

☞ a = { Leche }
Confidence = Support({Huevos, Leche}) / Support({Leche}) = 2/3
Regla **{Leche} ⇒ {Huevos} [2/3]**

❷ l_k = *{Huevos, Pan}*

GenerarReglas ({Huevos, Pan}, {Huevos, Pan})

A = { {Huevos}, {Pan} }

☞ a = { Huevos }
Confidence = Support({Huevos, Pan}) / Support({Huevos}) = 2/3
Regla **{Huevos} ⇒ {Pan} [2/3]**
☞ a = { Pan }
Confidence = Support({Huevos, Pan}) / Support({Pan}) = 2/3
Regla **{Pan} ⇒ {Huevos} [2/3]**

❸ l_k = *{Huevos, Patatas}*

GenerarReglas ({Huevos, Patatas}, {Huevos, Patatas})

A = { {Huevos}, {Patatas} }

☞ a = { Huevos }
Confidence = Support({Huevos, Patatas}) / Support({Huevos}) = 2/3
Regla **{Huevos} ⇒ {Patatas} [2/3]**
☞ a = { Patatas }
Confidence = Support({Huevos, Patatas}) / Support({Patatas}) = 2/2
Regla **{Patatas} ⇒ {Huevos} [2/2]**

❹ l_k = *{Leche, Pan}*

GenerarReglas ({Leche, Pan}, {Leche, Pan})

A = { {Leche}, {Pan} }

☞ a = { Leche }
Confidence = Support({Leche, Pan}) / Support({Leche}) = 3/3
Regla **{Leche} ⇒ {Pan} [3/3]**
☞ a = { Pan }
Confidence = Support({Leche, Pan}) / Support({Pan}) = 3/3
Regla **{Pan} ⇒ {Leche} [3/3]**

❺ l_k = *{Huevos, Leche, Pan}*

GenerarReglas ({Huevos, Leche, Pan}, {Huevos, Leche, Pan})

A = { {Huevos, Leche}, {Huevos, Pan}, {Leche, Pan} }

☞ a = { Huevos, Leche }
Confidence = Support(l_k) / Support(a) = 2/2
Regla **{Huevos, Leche} ⇒ {Pan} [2/2]**
GenerarReglas ({Huevos, Leche, Pan}, {Huevos, Leche})

A = { {Huevos}, {Leche} }

☞ a = { Huevos }
Confidence = Support(l_k) / Support(a) = 2/3
Regla **{Huevos} ⇒ {Leche, Pan} [2/3]**
☞ a = { Leche }
Confidence = Support(l_k) / Support(a) = 2/3
Regla **{Leche} ⇒ {Huevos, Pan} [2/3]**

☞ a = { Huevos, Pan }
Confidence = Support(l_k) / Support(a) = 2/2
Regla **{Huevos, Pan} ⇒ {Leche} [2/2]**
GenerarReglas ({Huevos, Leche, Pan}, {Huevos, Pan})

A = { {Huevos}, {Pan} }

☞ a = { Huevos }
Confidence = Support(l_k) / Support(a) = 2/3
Regla **{Huevos} ⇒ {Leche, Pan} [2/3]** *REPETIDA*
☞ a = { Pan }
Confidence = Support(l_k) / Support(a) = 2/3
Regla **{Pan} ⇒ {Huevos, Leche} [2/3]**

☞ a = { Leche, Pan }
Confidence = Support(l_k) / Support(a) = 2/3
Regla **{Leche, Pan} ⇒ {Huevos} [2/3]**
GenerarReglas ({Huevos, Leche, Pan}, {Leche, Pan})

A = { {Leche}, {Pan} }

☞ a = { Leche }
Confidence = Support(l_k) / Support(a) = 2/3
Regla **{Leche} ⇒ {Huevos, Pan} [2/3]** *REPETIDA*
☞ a = { Pan }
Confidence = Support(l_k) / Support(a) = 2/3
Regla **{Pan} ⇒ {Huevos, Leche} [2/3]** *REPETIDA*

4.2 Algoritmo mejorado

Ya se ha mostrado que, si la regla $s \Rightarrow (i-s)$ no se cumple con la fiabilidad mínima, tampoco alcanzarán *MinConfidence* las reglas de la forma $s' \Rightarrow (i-s')$ donde $s' \subset s$. Por lo tanto, para que se cumpla una regla $(i-c) \Rightarrow c$, todas las reglas de asociación de la forma $(i-c') \Rightarrow c'$ deben cumplirse, donde c' es un subconjunto no vacío de c.

Por ejemplo, si se verifica la regla $AB \Rightarrow CD$, necesariamente se verificarán las reglas $ABC \Rightarrow D$ y $ABD \Rightarrow C$. Aprovechando esta peculiaridad de las reglas de asociación se puede obtener un algoritmo más rápido para generar las reglas de asociación a partir de los itemsets relevantes.

Dado un itemset relevante l, si una regla de asociación derivada de l con c en el consecuente alcanza la fiabilidad mínima, también serán lo suficientemente fiables todas aquellas reglas de asociación con subconjuntos propios de c en el consecuente. Esta propiedad es similar a la que nos asegura que si un itemset es relevante también lo serán todos sus subconjuntos.

A partir de un itemset relevante l se pueden obtener todas las reglas de asociación derivadas de l con un único ítem en el consecuente que alcancen *MinConfidence*. A partir de estos items individuales se puede utilizar la misma función que se utilizaba para generar candidatos en el algoritmo *Apriori* y obtener de esa forma los posibles consecuentes de las reglas con dos items en el consecuente.

El algoritmo expuesto es una variación del expuesto por Agrawal y Skirant en su artículo *"Fast Algorithms for Mining Association Rules"*.

Algoritmo mejorado

```
H_n = { posibles n-itemsets consecuentes de reglas derivadas de l_k }

Para cada itemset relevante l_k (k≥2)
      H_1 = ∅
      Para cada item i de l_k
            confidence = support(l_k) / support (l_k-i)
            Si confidence > MinConfidence
                  Regla (l_k-i) ⇒ i [confidence, support(l_k)]
                  H_1 = H_1 ∪ {i}
      GenerarReglasApriori (l_k, H_1)
```

Algoritmo mejorado (continuación)

procedimiento GenerarReglasApriori (l_k , H_m)

```
Si k > m+1
      H_{m+1} = { Candidatos generados a partir de H_m (como C_{k+1} a partir de L_k) }
      Para cada h_{m+1} ∈ H_{m+1}
            confidence = support(l_k) / support (l_k - h_{m+1})
            Si confidence > MinConfidence
                  Regla (l_k - h_{m+1}) ⇒ h_{m-1} [confidence, support(l_k)]
            Si no
                  Eliminar h_{m+1} de H_{m+1}
      GenerarReglasApriori ( l_k , H_{m+1} )
```

Comparación de los dos algoritmos expuestos

Supongamos que tenemos un itemset relevante $l=\{A,B,C,D,E\}$ y que las reglas $ACDE \Rightarrow B$ y $ABCE \Rightarrow D$ son las únicas reglas de asociación derivadas de l con un ítem en el consecuente que alcanzan la fiabilidad mínima *MinConfidence*.

Si utilizamos el algoritmo directo, se comprobará si la fiabilidad de las reglas $ACD \Rightarrow BE$, $ADE \Rightarrow BC$, $CDE \Rightarrow BA$ y $ACE \Rightarrow BD$ alcanzan el umbral establecido. Si nos fijamos, la primera de las reglas, $ACD \Rightarrow BE$, no puede alcanzar *MinConfidence* ya que $\{E\} \subset \{B,E\}$ y la regla de asociación $ABCD \Rightarrow E$ no tiene la fiabilidad requerida. Las reglas segunda y tercera tampoco pueden alcanzar el umbral por razones similares. Por lo tanto, el algoritmo estándar comprobará cuatro reglas de asociación con dos items en el consecuente cuando sabemos de antemano que las tres primeras no pueden llegar a verificarse con la fiabilidad requerida.

En realidad, la única regla de asociación derivada de l con dos items en el consecuente que podría llegar a tener una fiabilidad por encima de *MinConfidence* es $ACE \Rightarrow BD$. De hecho, se puede comprobar sin dificultad que $ACE \Rightarrow BD$ es la única regla que comprobará el algoritmo mejorado.

EJEMPLO: Hagamos uso por última vez más la base de datos del supermercado:

TID	Items
Ana	Leche, Pan
Juan	Bacon, Huevos, Patatas
Luis	Huevos, Leche, Pan, Patatas
María	Huevos, Leche, Pan

A partir de las transacciones realizadas obtenemos los itemsets relevantes siguiendo cualquiera de los algoritmos ya expuestos (desde AIS hasta DHP):

Itemset	Support
{Huevos}	3
{Leche}	3
{Pan}	3
{Patatas}	2
{Huevos, Leche}	2
{Huevos, Pan}	2
{Huevos, Patatas}	2
{Leche, Pan}	3
{Huevos, Leche, Pan}	2

Para obtener las catorce reglas de asociación derivadas de estos itemsets, el algoritmo se aplica a cada k-itemset del conjunto de itemsets relevantes que contenga, al menos, dos items:

❶ l_k = *{Huevos, Leche}*

$H_1 = \varnothing$
☞ i = { Huevos }
Confidence = Support({Huevos, Leche}) / Support({Leche}) = 2/3
Regla **{Leche} ⇒ {Huevos} [2/3]**
H_1 = {{Huevos}}
☞ i = { Leche }
Confidence = Support({Huevos, Leche}) / Support({Huevos}) = 2/3
Regla **{Huevos} ⇒ {Leche} [2/3]**
H_1 = {{Huevos},{Leche}}

GenerarReglasApriori (l_k , H_l)
k ↛ 1+1

❷ l_k = *{Huevos, Pan}*

$H_1 = \varnothing$
☞ i = { Huevos }
Confidence = Support({Huevos, Pan}) / Support({Pan}) = 2/3
Regla **{Pan} ⇒ {Huevos} [2/3]**
H_1 = {{Huevos}}
☞ i = { Pan }
Confidence = Support({Huevos, Pan}) / Support({Huevos}) = 2/3
Regla **{Huevos} ⇒ {Pan} [2/3]**
H_1 = {{Huevos},{Pan}}

GenerarReglasApriori (l_k , H_1)
k ⇽ 1+1

❸ l_k = *{Huevos, Patatas}*

$H_1 = \varnothing$
☞ i = { Huevos }
Confidence = Support({Huevos, Patatas}) / Support({Patatas}) = 2/2
Regla **{Patatas} ⇒ {Huevos} [2/2]**
H_1 = {{Huevos}}
☞ i = { Patatas }
Confidence = Support({Huevos, Patatas}) / Support({Huevos}) = 2/3
Regla **{Huevos} ⇒ {Patatas} [2/3]**
H_1 = {{Huevos},{Patatas}}

GenerarReglasApriori (l_k , H_1)
k ⇽ 1+1

❹ l_k = *{Leche, Pan}*

$H_1 = \varnothing$
☞ i = { Leche }
Confidence = Support({Leche, Pan}) / Support({Pan}) = 3/3
Regla **{Pan} ⇒ {Leche} [3/3]**
H_1 = {{Leche}}
☞ i = { Pan }
Confidence = Support({Leche, Pan}) / Support({Leche}) = 3/3
Regla **{Leche} ⇒ {Pan} [3/3]**
H_1 = {{Leche},{Pan}}

GenerarReglasApriori (l_k , H_1)
k ⇽ 1+1

❺ l_k = *{Huevos, Leche, Pan}*

$H_1 = \varnothing$
☞ i = { Huevos }
Confidence = Support(l_k) / Support(l_k - {Huevos}) = 2/3
Regla **{Leche, Pan} ⇒ {Huevos} [2/3]**
H_1 = {{Huevos}}
☞ i = { Leche }
Confidence = Support(l_k) / Support(l_k - {Leche}) = 2/2
Regla **{Huevos, Pan} ⇒ {Leche} [2/2]**
H_1 = {{Huevos},{Leche}}
☞ i = { Pan }
Confidence = Support(l_k) / Support(l_k - {Pan}) = 2/2
Regla **{Huevos, Leche} ⇒ {Pan} [2/2]**
H_1 = {{Huevos},{Leche},{Pan}}

GenerarReglasApriori (l_k , H_1)

H_2 = { {Huevos, Leche}, {Huevos, Pan}, {Leche, Pan} }
☞ i = { Huevos, Leche }
Confidence = Support(l_k) / Support(l_k - {Huevos, Leche}) = 2/3
Regla **{Pan} ⇒ {Huevos, Leche} [2/3]**
☞ i = { Huevos, Pan }
Confidence = Support(l_k) / Support(l_k - {Huevos, Pan}) = 2/3
Regla **{Leche} ⇒ {Huevos, Pan} [2/3]**
☞ i = { Leche, Pan }
Confidence = Support(l_k) / Support(l_k - {Pan}) = 2/3
Regla **{Huevos} ⇒ {Leche, Pan} [2/3]**

Se puede apreciar que este algoritmo es más eficiente que el anterior por dos motivos. En primer lugar, como ya se ha mencionado anteriormente, necesita explorar menos combinaciones para obtener las reglas de asociación. Además, no genera reglas de asociación duplicadas (lo que sí sucedía en el algoritmo anterior).

5. Referencias bibliográficas

Charu C. Aggarwal & Philip S. Yu
"A New Framework for Itemset Generation"
ACM SIGMOD PODS'97, 1997

En este artículo se realiza una crítica del modelo clásico empleado en la obtención de reglas de asociación y se propone sustituir los itemsets frecuentes por itemsets "fuertes" [*strongly collective itemsets*] para conseguir un proceso de obtención de reglas más eficiente y productivo. El modelo empleado trata de eliminar la generación de reglas espúreas (aquéllas que no aportan nada nuevo) y evitar la no obtención de reglas potencialmente interesantes (debida al valor de *MinSupport,* que, en ocasiones, ha de fijarse demasiado alto para evitar una explosión combinatoria en la generación de reglas).

Rakesh Agrawal & Kyuseok Shim
"Developing tightly-coupled applications on IBM DB2/CS Relational Database System: Methodology and Experience"
IBM Research Report. IBM Almaden Research Center, San Jose, California, USA, 1995

Se discuten implementaciones alternativas del algoritmo Apriori sobre DB2, el sistema de bases de datos relacionales de IBM. Se propone hacer que gran parte de los cálculos necesarios los realice el propio servidor de bases de datos para disminuir la comunicación necesaria entre la aplicación cliente y el servidor (los experimentos se realizaron ejecutando ambos en la misma máquina, por lo que la comunicación cliente-servidor involucraba además cambios de contexto).

Rakesh Agrawal & Ramakrishnan Skirant
"Fast Algorithms for Mining Association Rules"
IBM Research Report RJ9839
IBM Almaden Research Center, San Jose, California, USA, June 1994

Sin duda, uno de los artículos más importantes sobre el tema. En él se presentan los algoritmos **Apriori** y **AprioriTID**, así como su combinación **AprioriHybrid**, para la obtención de todas las reglas de asociación existentes en una base de datos de transacciones. En el artículo se muestra cómo esta familia de algoritmos mejora los algoritmos anteriores (AIS y SETM).

Rakesh Agrawal & John C. Shafer
"Parallel Mining of Association Rules"
IEEE Transactions on Knowledge and Data Engineering
December 1996

Se explica cómo se puede implementar el algoritmo Apriori en un multiprocesador sin memoria compartida (realizando la comunicación mediante paso de mensajes [MPI]), lo que es aplicable también a un cluster de estaciones de trabajo conectadas a través de una red de interconexión de alta capacidad. Se proponen tres alternativas: **Count Distribution**, **Data Distribution** y **Candidate Distribution**.

Roberto J. Bayardo Jr.
"Efficiently Mining Long Patterns from Databases"
ACM PODS'98 [Principles of Database Systems]
Seattle, Washington, USA, June 1998

En este artículo se propone una forma alternativa de enfrentarse al problema de obtener el conjunto de itemsets relevantes a la que se utiliza en los algoritmos derivados de *Apriori*. Los algoritmos del tipo de *Apriori* no son adecuados cuando hay k-itemsets relevantes con k elevado (todos sus subconjuntos [2^k] son, a su vez, relevantes y como tales son tratados). El algoritmo propuesto, **Max-Miner**, extrae eficientemente sólo aquéllos itemsets relevantes que no estén incluidos en otros itemsets relevantes..

David W. Cheung, Jiawei Han, Vincent T. Ng, Ada W. Fu & Wongjian Fu
"A Fast Distributed Algorithm for Mining Association Rules"
Proceedings of the International Conference on Parallel and Distributed Information Systems
Florida, December 1996

En este trabajo se estudian relaciones interesantes entre itemsets relevantes localmente y globalmente en bases de datos distribuidas. Se expone un algoritmo distribuido de obtención de reglas de asociación, **FDM** [*Fast Distributed Mining of association rules*], que supone una mejora respecto al algoritmo *CD* [*Count Distribution*] propuesto por Agrawal y Shafer. Para afinar la eficiencia de este algoritmo se proponen distintas variantes del algoritmo: *FDM-LP* [*FDM with Local Prunning*], *FDM-LUP* [*FDM with Local and Upper-Bound Prunning*] y *FDM-LPP* [*FDM with Local Prunning and Polling-Site-Prunning*].

David W. Cheung, Jiawei Han, Vincent T. Ng & C.Y. Wong
"Maintenance of Discovered Association Rules in Large Databases:
An Incremental Updating Technique"
1996 International Conference on Data Engineering, New Orleans, Louisiana, February 1996

En este artículo se propone un algoritmo denominado FUP [Fast Update] para la actualización de un conjunto de reglas de asociación cuando la base de datos de la que se extrajeron ha sido modificada (esto es, nuevas transacciones han sido realizadas).

David W. Cheung, Vincent T. Ng & Benjamin W. Tam
"Maintenance of Discovered Knowledge: A Case in Multi-level Association Rules"
2nd International Conference on KDD, Oregon, August 1996

Este otro artículo de los investigadores de Hong Kong propone un algoritmo mejorado para el mantenimiento de las reglas de asociación extraídas de una base de datos transaccional al que denominan FUP* (un refinamiento de FUP) así como una generalización suya, el algoritmo MLUp, que es notablemente mejor que el algoritmo ML-T2.

R. Feldman, A. Amir, Y. Auman, A. Zilberstien & H. Hirsh
"Incremental Algorithms for Association Generation"
En "KDD. Techniques and Applications", H. Lu et al. eds., World Scientific, 1997.

Aquí se proponen tres algoritmos para el mantenimiento de un conjunto de reglas de asociación cuando se añade información a la base de datos de la que se obtuvieron. El primero de ellos es trivial y consiste en mantener la relevancia de todos los itemsets, relevantes o no. Los otros dos (algoritmo de propagación y algoritmo Delta) son algo mejores y requieren menos recursos.

Jia Liang Han & Ashley W. Plank
"Background for Association Rules and Cost Estimate of Selected Mining Algorithms"
ACM CIKM'96 (International Conference on Information and Knowledge Management)
Rockville, Maryland, USA, 1996

En este artículo se intenta comparar, desde un punto de vista estadístico, el coste computacional de distintos algoritmos para la obtención de reglas de asociación (Apriori, AprioriTid, AprioriHybrid, OCD, SETM y DHP) así como estudiar su escalabilidad.

M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen & A.I. Verkamo
"Finding Interesting Rules from Large Sets of Discovered Association Rules"
ACM CIKM'94 (International Conference on Information and Knowledge Management)
Gaitherburg, Maryland, USA, November 1994

Se trata de resolver el problema de cómo extraer, de todas las reglas de asociación obtenidas de una gran base de datos, las reglas que de verdad son interesantes para el usuario. Para ello se clasifican los atributos y se utilizan patrones (expresiones regulares) para seleccionar subconjuntos de reglas. Así mismo, se propone visualizar las reglas acompañadas de gráficos de barras (o gráficos de Kiviat) en los que aparezcan relevancia, fiabilidad y el producto de ambas (a lo que denominan 'commonness'). Esta técnica permite encontrar reglas interesantes en un simple vistazo.

Arno J. Knobbe, Pieter W. Adriaans
"Analysing Binary Associations"
2nd International Conference on KDD, Oregon, August 1996

Este artículo, igual que el de Klemettinen y sus colaboradores, está dedicado a la interpretación de las reglas obtenidas y no a su obtención. Aquí se propone obtener únicamente reglas de la forma A⇒C (asociaciones binarias entre pares de items producidas utilizando distintas medidas de similaridad, no necesariamente equivalentes a la fiabilidad de las reglas de asociación), construir un grafo que relacione los distintos items involucrados en las asociaciones binarias y obtener su árbol generador minimal de forma que visualmente se consiga representar la mayor parte del conocimiento obtenido.

Heikki Mannila, Hannu Toivonen & A. Inkeri Verkamo
"Improved Methods for Finding Association Rules"
Department of Computer Science, University of Helsinki, C-1993-65
Helsinki, December 1993

A partir del algoritmo AIS de Agrawal, Imielinski y Swami, proponen algunas mejoras. Por ejemplo, calcular C[k+1] a partir de C[k]*C[k] en vez de L[k]*L[k] para reducir el número de pasadas necesario por la base de datos. Además, se propone podar a priori el conjunto de candidatos C[k+1] generado a partir de L[k]. Ésta es la base del algoritmo *Apriori*, desarrollado en IBM independientemente del trabajo de estos finlandeses.

Andreas Mueller
"Fast Sequential and Parallel Algorithms for Association Rule Mining: A Comparison"
Department of Computer Science, University of Maryland - College Park, CS-TR-3515
College Park, MD, August 1995

En esta tesis se estudian y proponen distintos algoritmos de extracción de reglas de asociación: **SEAR** (una variante de Apriori que utiliza como estructura de datos un árbol de "prefijos" equivalente a un árbol de enumeración de subconjuntos), **SPEAR** (una modificación de SEAR que particiona horizontalmente las bases de datos), **SPTID** (como SPEAR, pero derivado de AprioriTID) y **SPINC** (un algoritmo incremental derivado de SPEAR). Además, se estudia la paralelización de estos algoritmos utilizando paso de mensajes en una arquitectura IBM PS2 [POWERparallel System],un multiprocesador con memoria distribuida). Como ejemplo se exponen **PEAR** (la versión paralela de SEAR) y **PPAR** (la correspondiente a SPEAR).

Raymond T. Ng, Laks V.S. Lakshmanan, Jiawei Han & Alex Pang
"Exploratory Mining and Pruning Optimizations of Constrained Association Rules"
Proceedings of the ACM SIGMOD Conference on Management of Data, SIGMOD'98
ACM, 1998

Este artículo está dedicado a la extracción selectiva de reglas de asociación mediante la especificación de restricciones por parte del usuario a modo de consultas SQL (*CAQs: Constraint Association Queries*). Se presenta un algoritmo denominado **CAP** (*Constrained Apriori*) que incorpora el uso de restricciones en el proceso de obtención de itemsets relevantes.

Jong Soo Park, Ming-Syan Chen & Philip S. Yu
"An Effective Hash-Based Algorithm for Mining Association Rules"
Proceedings of the ACM SIGMOD Conference on Management of Data
San Jose, California, USA, May 1995

En esta ponencia se propone el algoritmo **DHP** para disminuir el número de itemsets candidatos generados en las primeras iteraciones por algoritmos derivados de Apriori. Además, el tamaño de la base de datos que se debe recorrer se va reduciendo paulatinamente en cada iteración del algoritmo (las transacciones en las que ya sabemos que no se encuentra ningún itemser relevante [frecuente] no hay por qué comprobarlas en las siguientes iteraciones).

Jong Soo Park, Ming-Syan Chen & Philip S. Yu
"Using a Hash-Based Method with Transaction Trimming for Mining Association Rules"
IEEE Transactions on Knowledge and Data Engineering
September/October 1997

El artículo básicamente es idéntico a la ponencia de los autores en ACM SIGMOD'95. Como novedad, se propone reducir el número de pasadas realizadas por la base de datos generando los candidatos C[k+1] a partir de C[k]*C[k] en vez de L[k]*L[k]. Esto es válido si el conjunto de k-itemsets candidatos es lo suficientemente reducido para que no se produzca una explosión combinatoria.

Jong Soo Park, Philip S. Yu & Ming-Syan Chen
"Mining Asociation Rules with Adjustable Accuracy"
Proceedings of the 6th International Conference on Information and Knowledge Management
Las Vegas, Nevada, USA, 1997

Park, Yu y Chen proponen los algoritmos **DS** [Direct Sampling] y **SH**, derivados de DHP, para obtener reglas de asociación con una precisión ajustable. Se prima la eficiencia de la extracción de reglas que pueden ser de utilidad frente a la precisión en la obtención de todas las reglas de asociación. Consentir resultados imprecisos permite acelerar el proceso de '*data mining*'.

Sunita Sarawagi, Shiby Thomas & Rakesh Agrawal
"Integrating Association Rule Mining with Relational Database Systems:
Alternatives and Implications"
IBM Research Report. IBM Almaden Research Center, San Jose, California, 1998

Se comentan distintas implementaciones del algoritmo Apriori haciendo uso de SQL. Las implementaciones realizadas con SQL-92 son demasiado lentas (no mejoran la implementación clásica de Apriori, usada por *Intelligent Miner*™ de IBM). Utilizando características no estándar de SQL se consiguen algunas mejoras en tiempo de ejecución. Los mejores resultados se obtienen volcando inicialmente la base de datos en un fichero con el formato adecuado para su manipulación

Ramakrishnan Skirant & Rakesh Agrawal:
"Mining Quantitative Association Rules in Large Relational Tables"
Proceedings of the ACM SIGMOD Conference on Management of Data, SIGMOD'96
ACM, 1996

En este artículo se plantea el problema de la extracción de reglas de asociación en bases de datos que contengan atributos numéricos y se propone la utilización del método de partición equitativa [*equi-depth partitioning*]. Para obtener las reglas de asociación se expone una variante del algoritmo Apriori que hace uso de una medida de interés para podar los conjuntos de candidatos *C[k]*.

Ramakrishnan Skirant, Quoc Vu & Rakesh Agrawal
"Mining Association Rules with Item Constraints"
KDD'97 [AAAI Workshop on Knowledge Discovery in Databases]
AAAI, 1997

El artículo se centra en la obtención rápida de una parte del conjunto total de reglas de asociación que se puede obtener de una base de datos, para lo cual se proponen tres algoritmos derivados de *Apriori*: *MultipleJoins*, *Reorder* y *Direct*. Para acelerar el proceso, en vez de usar las restricciones especificadas por el usuario durante el post-procesamiento (una vez que ya se hayan obtenido todas las reglas de asociación), éstas se integran en el propio algoritmo de obtención de reglas de asociación. Las restricciones son expresiones boolenas acerca de la presencia o ausencia de determinados items. Además, se permite la introducción de taxonomías sobre los items permitiendo la utilización de los predicados *"ancestro(ítem)"* y *"descendiente(ítem)"* en la especificación de las restricciones.