제32회 추계학술발표회 논문집 Vol. 32, No. 2(Ⅱ)

BAR: 비트맵 기반의 연관규칙 구현 및 최적화

김재명° 오기선 김동현 이상원
성균관대학교 정보통신공학부
{jam02°, kisunoh, tube, swlee}@skku.edu

BAR: Bitmap-based Association Rule - Implementation and its Optimizations

Jae-Myung Kim° Ki-Sun Oh Dong-Hyun Kim Sang-Won Lee
School of ICE, SungKyunKwan University

요 약

대표적인 데이터마이닝 문제중의 하나인 연관규칙 탐사에는 지금까지 Apriori 기반의 많은 알고리즘들이 개발되어 왔다. 본 논문에서는 비트맵을 이용한 Apriori 알고리즘 구현방안을 제시한다. 우선, 핵심연산인 비트맵 논리곱(Bitmap AND)과 비트 카운팅(bit-counting)을 컴퓨터 CPU의 고급 기술을 이용해서 효과적으로 구현할 수 있음을 보인다. 또한, 트랜잭션 데이터를 비트맵으로 표현함으로써, 기존 Apriori와는 달리, 비트맵 논리곱 연산을 획기적으로 줄일 수 있는 방법을 제시한다. BAR의 이러한 구현기법을 통해, Apriori 기반의 최신 구현 방법에 비해, 성능이 최대 30배 정도 향상됨을 보인다.

1 서 론

데이터마이닝이란 많은 데이터 가운데 숨겨져 있는 유용한 상관관계를 발견하는 것이다. 이러한 데이터마이닝의 알고리즘 중에서 연관규칙 마이닝 알고리즘이란 빈도가 높은 패턴이나 항목들간에 연관관계를 찾는 알고리즘을 말한다.

가장 대표적인 연관규칙 마이닝 알고리즘은 1994년에 발표된 Apriori[1] 알고리즘이다. 이는 휴리스틱 방법을 이용하여 빈발 항목집합을 생성할 때, 후보 항목집합의 생성을 효과적으로 줄여준다. 하지만 근본적으로 후보 항목집합이 생성될 때 마다 트랜잭션 데이터를 스캔 해야 한다. 근래에도 이러한 Apriori 기반의 성능 개선을 위한 방안들이 지속적으로 개발되어 왔다.

[2]에서는 후보 항목집합들을 생성하지 않고 두 번의 트랜잭션 데이터 스캔으로 빈발 항목집합들을 생성할 수 있는 FP-Growth 알고리즘이 제안 되었다. FP-Growth는 트랜잭션 항목들간에 중복되는 내용이 많은 경우 좋은 성능을 보여준다. 하지만 트랜잭션을 이루는 항목들이 서로 다르면 새로운 노드를 생성하기 때문에 메모리 공간과 실행 시간에 대한 효율이 떨어지는 단점이 있다. Apriori와 FP-Growth 방법은 데이터의 특성에 따라 성능이 다르기 때문에, 지금까지는 어느 방법이 모든 면에서 더 우수하다는 결론은 없다.

본 논문에서는 비트맵을 이용한 Apriori 알고리즘 구현방안을 제시한다. 우선, 트랜잭션 데이터를 비트맵으로 표현함으로써, 기존 Apriori와는 달리, 비트맵 논리곱 연산만으로 항목집합을 구하고, 그러한 논리곱 연산 횟수를 획기적으로 줄일 수 있는 방법을 제시한다. 이를 통해 위에서 언급했던 Apriori와 FP-Growth가 가지는 단점들을 보완하였다. 또한 비트맵 기반의 Apriori 알고리즘 구현의 핵심연산인 비트맵 논리곱(Bitmap AND)과 비트 카운팅(bit-counting)을 컴퓨터 CPU의 고급 기술인 SIMD(Single Instruction Multiple Data)[3][4]를 이용하여 비트맵 연산 효율을 높이는 방안을 제시하였다. 이러한 기법은 하드웨어의 고급 기술을 데이터베이스 구현에 활용하는 최근 연구경향 [7]과 같은 맥락이다. 이러한 구현기법을 통해, Apriori 기반의 최신 구현 방법에 비해, 성능이 최대 30배 이상 향상될 수 있음을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 Apriori 알고리즘을 비트맵 개념을 이용해서 구현할 수 있음을 보이고, 3장에서는 비트맵 기반의 Apriori 알고리즘의 최적 구현 기법을 기술한다. 4장에서 가장 빠른 Apriori 기반의 구현 및 FP-Growth 기반의 구현과 성능 비교를 통해 본 논문에서 제안하는 BAR 기법의 우수성을 보이고, 5장에서 결론과 향후 연구방향을 설명한다.

2 BAR: 비트맵 기반의 연관 규칙

본 저자들은 비트맵 개념을 연관규칙 탐사에 적용하는 BAR 기법을 [8]에서 제시하였다. 본 논문은 [8]의 구현 기법과 그 과정에서 성능 최적화에 주안점을 두기 때문에, 독자들이 BAR의 개념을 이해할 수 있을 정도로 간단히 소개하고자 한다. 자세한 내용은 [8]을 참고하기 바란다.

2.1 비트맵의 적용

BAR 알고리즘은 각각의 항목에 대한 트랜잭션 정보를 비트맵으로 표현한다. 그림1은 트랜잭션 데이터 D에 대한 BAR 알고리즘의 수행 예제이다. 트랜잭션 데이터로부터 데이터를 읽어와 C_1을 생성한다. 항목 a에 대한 비트맵을 TID의 상대적 위치에 따른 집합으로 표현하면, {0,2}가 되고, 빈도수는 그 집합의 카디널리티와 같다.

또한 항목 x의 비트맵을 `x.bits`로 표시하고, `bitCount()`를 비트 카운팅 함수로 정의하면, 항목집합 {a, b}에 대한 빈도수는 아래와 같다.

`bitCount(and(a.bit, b.bits))` = *항목집합{a, b}의 빈도수*

특정 k항목집합에 대해 k-1번의 `and()`함수 호출과 한 번의 `bitCount()`함수 호출을 통해 해당 항목집합의 빈도수를 알 수 있다. 하지만 k-1번의 `and()`함수는 각각의 항목집합마다 수행되므로, 매우 많은 수의 `and()`함수 호출이 필요하다. 본 논문에서는 이를 줄이기 위한 효율적인 방안을 제시하였다.

TID	Items
100	a c d
200	b c e
300	a b c e
400	b e

D

TID	{a}	{b}	{c}	{d}	{e}
100	1	0	1	1	0
200	0	1	1	0	1
300	1	1	1	0	1
400	0	1	0	0	1
supp	2	3	3	1	3

C_2

TID	{a}	{b}	{c}	{e}
100	1	0	1	0
200	0	1	1	1
300	1	1	1	1
400	0	1	0	1
supp	2	3	3	3

F_1

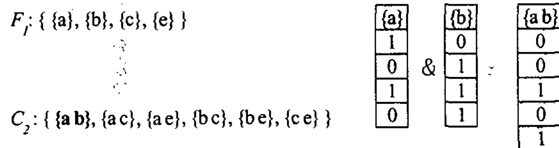

그림1 비트맵 적용 예제

2.2 BAR 알고리즘

BAR 알고리즘은 크게 트랜잭션 데이터를 읽고, 빈번한 항목들을 찾아서 2항목집합을 찾는 과정과 반복적으로 k항목집합을 찾는 과정으로 나눌 수 있다. k항목집합을 찾는 과정은 아래와 같이 3단계로 이루어 진다.

① gen_ck(): Apriori의 조인 및 가지치기 개념을 적용하여 이전 단계의 항목집합에서 얻은 정보를 통해 불필요한 후보 항목집합의 생성을 줄인다.

② bitmap(): gen_ck()에서 생성한 후보 항목집합의 비트맵을 생성하고, 빈도수를 계산한다. 이 과정에서 공통 비트맵 후보를 이용하여 AND함수 호출의 수를 줄인다.

③ take_fk(): 후보 k항목집합의 리스트에서 최소 빈도수보다 작은 후보 항목집합을 제거한다.

```
C1 ← gen_c1(D)
F1 ← take_f1(C1, minsup)
F2 ← gen_f2(F1, minsup)
FOR ( k ← 3; Fk-1≠∅; k++ ) DO
BEGIN
      Ck ← gen_ck(Fk-1)
      bitmap(Ck)
      Fk ← take_fk(Ck, minsup)
END
      Write result to file
```

그림2 BAR 알고리즘

3 구 현

본 논문에서는 논리적으로 공통 비트맵 후보를 유지하는 알고리즘을 적용하고, 구현의 측면에서 and()와 bitCount()함수에 대한 성능을 향상 시켰다. 또한 하드웨어 고급 기술을 활용하기 위하여 비트맵 연산에 SIMD를 적용하였다. 동적 비트맵의 표현과 비트맵 논리곱을 위해 Java SDK 1.5의 공개된 BitSet 클래스 구조를 참조하였다.

3.1 공통 비트맵 후보의 유지

공통 비트맵 후보란 특정 항목집합 생성 시 함께 생성된 항목집합의 부분집합과 부분집합의 비트맵을 의미한다.

공통 비트맵 후보를 유지하는 것은 k항목집합의 비트맵 생성 시 and()함수의 호출을 줄여준다. 특정한 k항목집합의 항목들은 이전의 k항목집합들과 공통된 항목을 가질 확률이 높다. 그 이유는 후보 항목집합을 생성하는 과정은 사전편찬순서 혹은 임의로 정한 항목들의 순서를 유지하며, 생성된 항목집합 또한 생성된 순서를 유지하고 있기 때문이다.

k-1항목집합을 조인하여 k항목집합을 생성할 때, 선위 k-2항목이 동일한 두 개의 k-1항목집합의 결합은 많은 k-1항목 이하의 공통된 항목들을 가진다. 우리는 공통된 항목에 대한 비트맵을 유지하는 알고리즘을 통해 성능을 개선하였다.

공개된 트랜잭션 데이터인 accidents.dat[10]를 최소 지지도 40%로 비교 수행하였을 때, 비트맵에 대한 and()함수의 호출 횟수가 159,760회 이었던 것이 51,393회로 약 70%가량 줄었다.

3.2 and()함수

BAR 알고리즘에서 사용하는 특정한 비트맵은 메모리 공간에 연속적으로 존재한다. 하지만 각각의 서로 다른 비트맵간의 연속성은 보장할 수 없고, 각각의 다른 비트맵의 크기 또한 서로 같다고 할 수 없다. 이는 메모리 공간의 효율성을 위해 비트맵의 크기를 동적으로 유지하기 때문이다.

비트맵 and()함수는 두 메모리 공간의 데이터를 읽어와 AND하고 그 결과를 다시 메모리 공간에 저장함을 의미한다.

이 때, 비트맵 크기가 동적이라는 점과 메모리 공간에 다시 저장해야 한다는 두 가지 특징으로 and()함수의 구현은 성능에 많은 영향을 준다.

and()함수의 두 가지 방법: ① a&=b; ② c=a&b;

예를 들어 ①번과 같이 계산 할 경우 각 항목의 원래 비트맵은 유지되어야 한다. a항목의 비트맵은 나중에 다른 항목집합의 비트맵을 생성하는데 계속 사용되어야 하기 때문이다. 결국 a&=b의 연산은 c=a; c&=b; 로 고쳐져야 한다. 또한 c&=b; 연산은 각각의 메모리 읽기 두 번과 c로의 메모리 쓰기 한 번이 발생하며, a의 사본을 유지하기 위한 메모리 읽기 쓰기가 각각 한번씩 추가로 발생한다.

반면 ②번의 경우 a와 b에 대한 메모리 읽기 두 번과 c에 대한 메모리 쓰기 한 번만으로 연산을 종료한다. 한마디로 불필요한 사본 생성에 대한 읽기 쓰기 연산이 절약된다.

뿐만 아니라 추가적인 이익도 얻을 수 있다. AND연산의 특성과 동적으로 구현한 비트맵 덕분이다. AND연산은 다음과 같은 식을 만족시킨다. $A \wedge 0 = 0$ (식1)

예를 들어 비트맵 a의 길이는 4096이고, 비트맵 b의 길이가 1024이라고 가정하자. 길이가 1024라는 것은 최대 1023번 비트까지 1이고 나머지는 모두 0이라는 의미이다. 다시 말하면 a&b의 결과에서 1024 ~ 4095번 비트는 모두 0임을 알 수 있다. 이를 이용하여 비트맵c는 처음부터 1024비트 만큼의 메모리만 할당하면 되고 AND연산 또한 그 만큼만 하게 된다. 예제의 경우에는 메모리 공간과 AND연산 횟수의 75%가 절약된다.

3.3 비트 카운팅

비트 카운팅이란 비트맵 내의 참인 비트 개수를 세는 것 이다. 비트 카운팅 알고리즘은 여러 가지가 있다. 가장 쉽게 생각할 수 있는 알고리즘은 1bit씩 비트의 값이 참인지 확인 하는 방법이지만, 이는 비트맵의 길이만큼 확인해야 하므로 효율이 떨어진다.

구현 첫 단계에서는 워드당 비트 수에 로그비율의 성능을 보이는 알고리즘[5]을 적용하였다. 이 알고리즘은 한 워드당 약 16번 정도의 연산이 필요하다.(방법1)

카디널리티 계산 알고리즘은 BAR에서 많은 비중을 차지하는 연산이기에 좀 더 빠른 방식의 연산이 필요하였다. 8비트 단위의 룩업 테이블을 이용하여, 참조하며 계산하는 방식을 적용하여 성능향상 효과를 얻었다.[5](방법2)

bitCount()함수를 호출 하는 시기 또한 성능에 영향을 미친다. 3.2에서 and()함수를 호출한 직후에 카디널리티를 계산하면, 비트맵의 크기가 L2캐시보다 작은 경우 and()함수의 결과인 비트맵이 L2캐시에 남아있기 때문에 메모리에서 L2캐시로의 추가적인 전송이 일어나지 않아 성능이 향상된다.

3.4 SIMD의 적용

SIMD(Single Instruction Multiple Data)는 한번의 명령어로 여러 개의 데이터를 처리하는 방식이다. 본 논문에서는 Intel® Pentium®4 processor에서 제공하는 SSE2(Streaming SIMD Extensions 2)를 이용하여 and()함수와 bitCount()함수를 SIMD화 하였고, 이를 통한 성능향상을 보였다. (방법2)의 경우 SIMD화 하기 어려운 부분이 있어 (방법1)을 SIMD화 하였다.

공개된 트랜잭션 데이터인 accidents.dat[10]를 최소 지지도 40%로 비교 수행하였을 때, 51,393번의 and()함수 수행시간이 5.57초에서 1.47초로 줄었다.

4 성능평가

공개된 Apriori와 FP-Growth 알고리즘의 구현 중에서 좋은 성능으로 평가 받고 있는 Borgelt의 구현[11][12]과 BAR를 비교하고, 이전 버전의 BAR 구현과 최적화된 BAR의 성능을 비교하였다. 하드웨어는 Intel® Pentium®4 processor 2.0 GHz, 512KB L2캐시, 512MB 메인 메모리, 7200rpm 2MB캐시의 하드디스크를 사용하였고, 운영체제는 Windows Server 2000 SP4, 컴파일러는 Intel® C++ Compiler 9.0을 사용하였다. 공정한 평가를 위하여 비교 대상은 모두 최적화 컴파일 하였다.

데이터는 [10]에서 다운로드 받았으며, 최소 지지도를 변경하며 반복 실험하였다. 가로축은 최소 지지도(%)를 나타내고 세로축은 수행시간(초)을 나타낸다.

그림3 에서는 BAR와 [11][12]의 성능을 비교하였다.

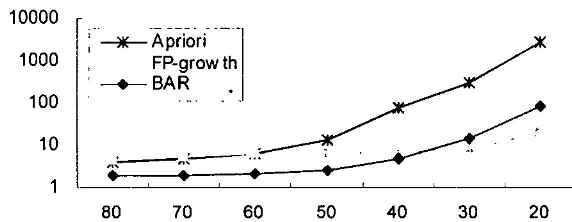

그림3 accidents.dat

그림4 에서는 BAR 알고리즘과 이전 버전 간의 구현방식에 따른 성능을 비교하였다. BAR_Init은 초기버전이며, BAR_CBC는 공통비트맵 후보 유지 알고리즘을 적용하고, 그에 따른 AND연산의 효율을 향상 시킨 버전이다. BAR_SIMD는 최종적으로 SIMD까지 적용된 버전이다.

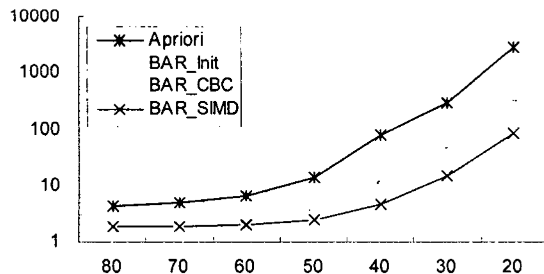

그림4 accidents.dat

5 결 론

Apriori 알고리즘의 가장 큰 단점은 빈발 항목집합의 숫자가 늘어나면 수행시간도 선형적으로 비례하여 증가한다는 것이다. 반면 FP-Growth는 트랜잭션을 이루는 항목들의 중복이 많으면 결과가 되는 빈발 항목집합의 수에 영향을 적게 받는다. 반면 트랜잭션을 이루는 항목들의 중복이 적으면 새로운 노드를 생성하고 관리하는 비용이 증가한다. 그러한 이유로 특정 데이터에 대해 좋지 못한 성능을 보이는 경우가 있으며, [6]에서는 FP-Growth가 실제 데이터에 대해서는 오히려 성능이 좋지 않다고 한다.

BAR 알고리즘은 기본적으로 Apriori에 기반하고 있으며, 추가적으로 트랜잭션간의 중복을 활용하기 위해 공통 비트맵 후보를 이용한다. 이를 통해 많은 빈발 항목집합을 생성할 때에도 로그 비례에 가까운 성능을 보인다.

결과적으로 [12]에 비해서는 항상 좋은 성능을 보여주고 있으며, [11]에 비해서는 최소 지지도 40%까지는 더욱 좋은 성능을 보여준다. 이는 최소 지지도가 40%일 때 약 15만 건, 30%일 때 약 90만 건의 빈발 항목집합을 생성함에 따른 성능 저하이다. 이는 Apriori와 FP-Growth와의 알고리즘적인 차이 때문인데, Apriori의 경우 낮은 항목집합에서 높은 항목집합순서로 정보를 찾아나가지만, FP-Growth의 경우 공통되는 트랜잭션 항목 순서로 찾고 바로 결과를 파일로 쓰기 때문이다. 이처럼 많은 빈발 항목집합이 생성되면 빈발 항목집합에 대한 정보로서의 가치는 떨어지게 되므로 실제로 이러한 쿼리가 가지는 의미는 크지 않다. [6] 또한, 로그 그래프임을 감안할 때, BAR는 월등한 성능을 보여준다.

향후 연관규칙 마이닝에 사용되는 실제 데이터는 트랜잭션의 수는 많고, 빈번 항목집합의 수는 적다는 사실에 착안하여 비트맵의 연산 효율을 향상 시키기 위해 압축기법을 적용하고, 비트맵의 특성을 효과적으로 이용하기 위한 다양한 방안 연구를 계속할 계획이다.

참고문헌

[1] R. Agrawal and R. Srikant. "Fast algorithms for mining association rules in large databases" In VLDB, pages 487-499, 1994.

[2] J. Han, J. Pei and Y. Yin. "Minining frequent patterns without candidate generation" In ACM SIGMOD, pages 1-12, 2000.

[3] Jingren Zhou, Kenneth A. Ross, "Implementing database operations using SIMD instructions" Proceedings of the SIGMOD, 2001

[4] "IA-32 Intel® Architecture Optimization Reference Manual" Intel Corporation, 2005.

[5] J. Nievergelt and K. Hinrichs. "Algorithms and data structures with applications to graphics and geometry" Prentice Hall, 1993.

[6] Zijian Zheng and Ron Kohavi and Llew Mason. "Real world performance of association rule algorithms" In ACM SIGKDD, pages 401-406, 2001.

[7] Gustavo Alonso, "Database for New Hardware" IEEE Data Engineering Bulletine, 2005, June.

[8] 김동현, 강운학, 이상원 "MBAR: 실체화된 비트맵 기반의 연관 규칙 알고리즘" 삼성 휴먼테크 2004 논문상 장려상 수상, 한국정보과학회 데이터베이스연구 20권 4호, 2004.

[9] 김재명, 오기선, 김동현, 이상원 "BAR: 비트맵 기반의 연관규칙 구현 및 최적화 (Long Version)"

[10] Frequent Itemset Mining Implementations Repository - http://fimi.cs.helsinki.fi/

[11] Christian Borgelt, "An Implementation of the FP-growth Algorithm" OSDM'05, Chicago, IL

[12] Christian Borgelt, "Recursion Pruning for the Apriori Algorithm" FIMI 2004, Brighton, UK