L2-tree를 이용한 효율적인 빈발항목 집합 탐사

박인창[0] 장중혁 이원석
연세대학교 컴퓨터과학과
{icpark[0], jhchang, leewo}@amadeus.yonsei.ac.kr

An Efficient Algorithm for mining frequent itemsets using L2-tree

In-Chang Park[0] Joong-Hyuk Chang Won-Suk Lee
Dept. of computer science, Yonsei University

요 약

데이터마이닝 분야에서 빈발항목집합 탐사에 관한 연구는 활발히 진행되어 왔지만 여전히 많은 메모리 공간과 시간을 필요로 한다. 특히 apriori 알고리즘에 기반한 방법들은 긴 패턴이 생성될수록 지수적으로 시간과 공간이 증가한다. 최근에 발표된 fp-growth는 일반적인 데이터 집합에서 우수한 성능을 보이나 희소 데이터 집합에서 효율적인 성능을 보여주지 못한다. 본 논문에서는 길이가 2인 빈발항목집합 L2에 기반한 L2-tree 구조를 제안한다. 또한 L2-tree에서 빈발항목집합을 탐사하는 L2-traverse 알고리즘을 제안한다. L2-tree는 L2를 기반으로 하기 때문에 L2가 상대적으로 적은 희소 데이터 집합 환경에서 적은 메모리 공간을 사용하게 된다. L2-traverse 알고리즘은 별도의 추출 데이터베이스를 생성하는 FP-growth와 달리 단순히 L2-tree를 오직 한번의 깊이 우선 탐사를 통해 빈발항목집합을 찾는다. 최적화 기법으로써 길이가 3인 빈발항목집합 L3가 되지 않는 L2 패턴들을 미리 제거하는 방법으로 C3-traverse 알고리즘을 제안하며 실험을 통해 기존 알고리즘과 비교 검증한다.

1. 서 론

데이터 마이닝은 데이터 집합을 분석하여 내재된 지식을 추출하는 것을 주된 목표로 하고 있다. 대표적인 데이터마이닝 분야인 연관규칙 탐색에서 연관규칙은 $X \Rightarrow Y$로 표현되며 X는 조건부, Y는 결과부라 하고 $X \cap Y = \varnothing$을 만족한다. 이 규칙의 의미는 매우 직관적인데, 트랜잭션의 데이터베이스 D가 주어지고, 각각의 트랜잭션 $T \in D$는 항목의 집합일 때, "X를 포함하는 트랜잭션은 Y를 포함하고 있을 것이다" 라는 의미를 가진다. 연관규칙은 지지도 와 신뢰도를 이용하여 규칙의 강도를 정한다. 지지도는 $|X \cup Y|$/전체트랜잭션의 개수로 표현되며 전체 데이터베이스에서 X와 Y가 차지하는 비중을 의미한다. 신뢰도는 $|X \cup Y|/|X|$로 표현되며 조건부 확률 $p(X \cup Y|X)$와 같다. 신뢰도가 의미하는 것은 항목 X가 제시되었을 때 항목 Y가 나타날 확률이다.

연관규칙은 알고리즘 수준에서 크게 두 가지의 처리단계를 거치게 된다. 하나는 빈발항목집합(frequent itemset)을 찾는 것이고, 다른 하나는 빈발항목집합에서 규칙을 생성하는 것이다. 연관 규칙 탐사의 전체 성능은 사실상 첫 번째 단계에서 결정된다. 그러므로 기존의 연관 규칙에 대해 발표된 많은 논문들이 방대한 시간이 소모되는 빈발항목집합을 찾는 첫 번째 단계에 중점이 두어졌다. 빈발항목규칙 탐색과 관련하여 가장 대표적인 알고리즘은 Apriori[1]이다. Apriori는 빈발 k-1 항목집합을 바탕으로 k항목집합을 생성하며 빈발하지 않는 k-1 항목집합을 전지 한다. AprioriTID[1]는 트랜잭션들의 전처리 과정을 통해 생성된 후보항목을 포함하고 있는 TID 리스트를 사용하여 표현하는 방법이다. AprioriHybrid[1]는 Apriori와 AprioriTID의 방법을 병합한 것이다. Partition[2] 알고리즘은 Apriori와 유사한 알고리즘으로 크게 다른 점은 지지도를 계산하기 위해서 항목집합의 TID 리스트간에 교집합 연산을 사용했다. 이러한 apriori 알고리즘에 기반한 방법들의 공통된 단점은 많은 후보 항목의 생성과 후보 항목에 대한 출연 빈도 갱신 비용이 크며 결국 주된 비용은 데이터베이스 스캔 회수보다는 후보 항목 생성에 있다. 비교적 최근에 발표된 FP-growth[3]와 TreeProjection[4] 방법은 패턴-성장(pattern-growth) 방법을 사용한 것으로 후보항목을 생성하지 않고 발견된 빈발항목을 기반으로 하는 추출 데이터베이스를 만들어 패턴의 길이가 하나 늘어난 빈발항목집합을 찾는 방법이다. 이 방법은 기존의 apriori 기반 방법보다 시간, 공간적으로 효율적인 알고리즘이며 주된 이유는 후보항목을 생성하지 않는데 있다. 그러나 FP-growth는 희소 데이터 집합에서 효율적이지 못한 성능을 보인다[5]. 주된 이유는 FP-growth는 크기가 1인 빈발항목집합 L1을 구하고 L1에 해당하는 모든 데이터를 메모리에 FP-tree[3] 구조로 구축되며, 데이터 집합의 밀도가 클수록 공유되는 부분이 적어지고 요구되는 메모리 크기도 매우 커진다. 특히 FP-growth 에서 조건패턴에 기반한 추출 데이터베이스인 조건 FP-tree를 생성하게 되는데 이것의 메모리 요구양도 역시 커진다. 본 논문에서 이러한 문제점을 해결하기 위해서 L2를 배열을 통해 빠르게 구한 후 L2-tree를 구성, 빈발항목집합을 탐사한다. 일반적으로 희소데이터 집합은 L2의 크기가 상대적으로 적고 메모리 요구량도 적어진다. 특히 L3가 될 수 없는 L2를 트리에 반영하지 않는 보다 효율적인 알고리즘인 C3-traverse를 제안한다.

2. L2-Tree 요소와 구성

먼저 데이터베이스의 첫 번째 스캔을 통해 빈발항목 L1을 구한다. 이를 바탕으로 두 번째 스캔을 통해 길이가 2인 빈발항목집합 L2를 $|L1|*|L1|$ 크기의 배열을 사용하여 빠르게 구할 수 있다.

표 1 트랜잭션 데이터베이스

TID	구입한 항목	빈발 항목	지지도 내림차순
100	*a, d, e*	*a, d, e*	*a, d, e*
200	*a, c, h, i*	*a, c, h*	*c, a*
300	*a, b, g, j*	*a, b, g*	*a, b, g*
400	*b, c, d, e, g*	*b, c, d, e, g*	*c, b, d, e, g*
500	*c, d, e, f*	*c, d, e*	*c, d, e*
600	*b, c, g*	*b, c, g*	*c, b, g*

표1의 트랜잭션 데이터베이스에서 최소지지도가 2 일 때 L1은 *c, a, b, d, e, g*가 되며 각각의 출연빈도 수는 4, 3, 3, 3,

3, 3이다. 구입한 항목에서 L1만을 만족하는 빈발항목을 추출하고 항목의 순서를 지지도 내림차순으로 나열한다. 본 논문에서 제안하는 L2-tree는 빈발항목집합의 크기가 2인 L2를 기반으로 트리를 생성한다. L2를 구하면 *cb, cd, ce, cg, bg, de* 이며 출현빈도는 2, 2, 2, 2, 3, 3이다.

표 2 L2-tree에 적용될 L2 패턴 생성

TID	지지도 내림차순 항목	만족하는 L2 집합	Tree에 입력될 L2-패턴
100	*a, d, e*	*(de)*	*d, e*
200	*c, a*	·	·
300	*a, b, g*	*(bg)*	*b, g*
400	*c, b, d, e, g*	*(cb)(cd)(ce)(cg)* · ·	*c, b, d, e, g* *b, g* *d, e*
500	*c, d, e*	*(cd) (ce)* ·	*c, d, e* *d, e*
600	*c, b, g*	*(cb) (cg)* ·	*c, b, g* *b, g*

L2를 만족하는 트랜잭션은 각 항목을 시작으로 하는 L2-패턴을 생성한다. 예를 들어 TID 400의 *c,b,d,e,g*는 각각의 항목으로 시작하는 *cbdeg, bdeg, deg, eg, g*를 생성 가능하다. 그러나 *bdeg*에서 *bg*만이 L2를 만족하고 deg는 de만이 L2를 만족한다. eg, g는 입력할 필요가 없는데, 이유는 지지도 내림차순의 마지막 2개의 항목은 최대 L2이기 때문에 L3이상을 위해 생성되는 L2-tree에 적용할 필요가 없다. 같은 이유로 길이 2 이하의 패턴 *bg, de*를 생성할 필요가 없다. 따라서 *c,b,d,e,g*는 *cbdeg*를 생성한다. 트랜잭션 T_k에서 L1을 만족하는 항목 집합을 T_k'이라 하고 항목의 개수를 $|T_k'|$라 하면 $T_k' = \{ i_1, i_2, \ldots, i_{|T_k'|} \}$와 같이 나타낼 수 있다. $S(i_n)$이 i_n의 출연빈도를 나타낼 때, 정수 σ, τ가 $\sigma > \tau$이면 $S(i_\sigma) < S(i_\tau)$를 만족한다. L2는 $i_\alpha i_\beta$로 구성되며 $\alpha < \beta$를 만족한다. 전자항목 α일 때 만족되는 모든 후자항목의 집합을 후자항목집합을 γ_α로 표현할 때, 항목 e로 시작하며 T_k'에서 생성될 수 있는 L2-패턴 $R(e, T_k')$는 식(1)과 같다.

$$R(e, T_k') = \begin{cases} \{e\} \cup (\gamma_e \cap T_k'), & \text{if } |\gamma_e \cap T_k'| \geq 2 \\ \emptyset, & otherwise \end{cases} \quad \text{식(1)}$$

T_k'에서 L2-tree에 입력되는 L2-패턴 집합 $\psi(T_k')$는 식(2)과 같다.

$$\psi(T_k') = \{x : x \in R(a_n, T_k'),\ n = 1, 2, \ldots, |T_k'| - 2\} \quad \text{식(2)}$$

L2-tree는 깊이 레벨 1의 항목별로 header table이 존재한다. header table a는 a로 시작하는 모든 경로의 깊이 레벨 3 이상의 노드들을 항목별로 연결한다. 이를 항목연결이라 한다. 항목연결은 깊이 레벨의 내림차순으로 연결된다. 예를 들어 *abd*와 *abcd*가 연속적으로 들어갈 때 a 헤더 테이블의 d 항목은 트리에 입력된 순서가 아니라 *(abcd)→(abd)* 순서가 된다. 항목 연결의 정렬은 노드 생성과 동시에 처리될 수 있다. L2-tree의 노드는 count와 addedCount항목을 가진다. count 항목은 트리가 구성될 때 트랜잭션의 입력에 의해서 더해지는 출연 빈도 값이다. addedCount는 같은 항목연결에 연결되어 있으면서 항목집합 e의 super set을 만족하는 모든 항목의 출연 빈도를 더해준 값이다. 빈발항목집합으로 판정될 출연 빈도는 count + addedCount이다. 예를 들어 *abf, abcf, abcdf, adcf* 가 들어왔을 때 *(abf)*의 출연 빈도를 구하는 것은 다음과 같다. header table a의 *f*항목연결은 *(abcdf)→(abcf)→(adcf)→(abf)* 가 된다. 이때 *(abf)*의 super set을 만족하는 노드는 *(abcdf),(abcf)*이다. 그러므로 *(abf)*. addedCount는 *(abcdf)*.count + *(abcf)*.count가 되며 결국 *(abf)*의 출연빈도는 *(abcdf)*.count + *(abcf)*.count + *(abf)*.count가 된다. addCount는 L2-tree가 구성된 후 레벨 3 이상의 노드들에 대해서 한번에 갱신한다. 그림1은 표1, 표2를 바탕으로 구성된 L2-tree이며 노드 레이블의 항목명(C)의 C는 출현 빈도를 나타낸다.

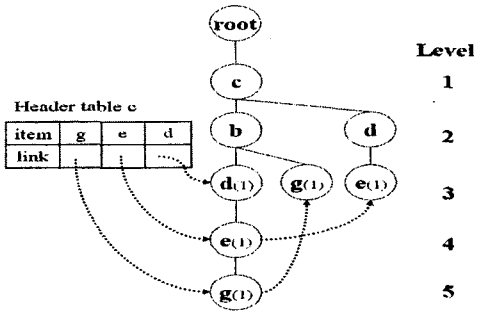

그림 1 L_2-tree

3. 빈발항목집합 탐사

3.1 L2-Traverse 알고리즘

L2-Traverse는 fp-growth와 달리 별도의 다른 구성없이 L2-tree를 한번 순회하면서 빈발항목집합을 탐사한다. 그림1을 바탕으로 예를 들어 설명하도록 한다. 먼저 루트 노드에서 c를 방문한다. 항목 *c* 이후에 나타날 수 있는 항목은 *c*보다 출현빈도가 작고 최소지지도보다 큰 항목이다. 그러므로 *a, b, d, e, g*가 올 수 있다. L2-tree가 일반적인 트리 순회와 다른 점은 tree상의 노드가 존재하지 않더라도 가능한 패턴의 항목에 대한 가상노드(virtual node)를 접근해야 한다. 예를 들어 *(ca)* 노드는 없지만 상황에 따라 가상노드로써 접근 할 수 있다. 가상노드의 출현빈도는 count가 존재하지 않기 때문에 addCount만으로 빈발항목여부를 판단한다. 또한 가상노드의 addCount는 미리 값을 구할 수 없기 때문에 가상 노드를 방문한 후 구한다. 만약 어떤 노드가 가상노드라면 그 이하의 모든 자식 노드들은 가상노드로써 접근한다. 그림 1을 참고하면 *(c)*를 방문한 후에 *(ca)*를 방문해한다. 하지만 *(ca)*는 존재하지 않으며, 가상노드로 방문할 필요가 없는 이유는 L2가 되가 되지 않기 때문이다. 이것은 *(ca)*로 시작하는 L3가 존재하지 않는다는 것을 의미한다. 따라서 *(ca)*라는 가상 노드를 방문할 필요가 없다. 즉 가상 노드는 L2를 만족하면서 노드가 tree에 존재하지 않을 때 방문한다. (ca)아래 노드를 순회할 필요가 없음으로 다음으로 방문할 노드 *(cb)*로 이동하면 그림 1에서와 같이 메모리 상에 존재하는 노드이다. L2를 기반으로 구축된 L2-tree에서는 깊이 레벨 2이하에 존재하는 모든 노드는 빈발항목집합이 된다. 그러므로 *(cb)*는 빈발항목이 된다. *(cb)*에서 가능한 항목인 *d, e, g*이다. *(cbd)*는 실제 노드이며 *(cbd)*.count와 *(cbd)*. addedCount를 더하여도 최소지지도를 넘지 않음으로 *(cbd)*보다 깊은 탐색을 할 필요가 없다. *(cbe)*는 실제로 존재하지 않는 가상 노드이다. 따라서 *(cbe)*.count의 값은 0이 됨으로 *(cbe)*.addedCount을 구하여 빈발 여부를 판단한다. *(cbe)*.addedCount는 같은 항목 연결에 있으면서 *(cbe)*의 super set을 만족해야 한다. 만족하는 노드는 *(cbde)*가 되고 결국 *(cbe)*.count는 1이 된다. 최소지지도를 만족하지 못하는 *(cbe)*는 빈발항목집합이 되지 않고 *(cbe)*를 선행경로로 가지는 모든 경로에 대해서 깊이 탐색이 필요 없다. 재귀적으로 *(cb)*로 돌아오게 되고 처리되지 않은 가능한 항목인 g를 탐사한다. *(cbg)*.count = *(cbg)*.count + *(cbg)*.addedCount, 값은 2이며 최소 지지도를 만족함으로 *(cbg)*는 빈발항목집합이다. 이런 방

식으로 L2-tree를 순회하면서 모든 L3이상의 빈발항목집합을 구한다.

3.2 C3-Traverse 알고리즘

L2를 만족하는 패턴은 L2-tree에 반영된다. 하지만 L3가 전혀 되지 않는 L2도 존재한다. 이것에 초점을 맞추는 이유는 L2-tree를 구성하는 이유가 L3 이상을 구하는 단계이기 때문이다. 본 논문에서 제안하는 최적화 방법을 설명하기 위해서 표3의 배열구조에 L2 count가 기록되어 있으며 최소지지도는 5로 가정한다.

표 3 L2-배열

항목	*b*	*c*	*d*	*e*
a	6	5	4	7
b		2	6	4
c			4	1
d				4

표3를 참조하면 *ab*의 출연 회수는 6이므로 빈발항목집합이며 L2-tree에 반영되어야 한다. *ab*를 시작으로 하는 경로는 *abc, abd, abe*가 된다. *abc*가 빈발항목집합이 되기 위해서는 *ab, bc, ac*가 빈발항목 집합이 되어야 하며 *bc*는 2이고 *ac*는 5이다. *bc*가 빈발패턴이 아니므로 *abc*는 빈발항목집합이 아니다. 비슷하게 *abd*의 경우 *ad, abe*의 경우 *be*가 빈발 항목집합이 아니므로 빈발항목집합이 아니다. 결국 *ab*로 시작하는 L3는 존재하지 않으며 *ab*를 L2-tree에 반영할 필요가 없다. 최적화를 위한 패턴 길이 2의 $\alpha\beta$ 출연 빈도 $C(\alpha,\beta)$는 식(3)과 같다.

$$C(\alpha,\beta)=\begin{cases} -1, \text{ if } R(\alpha,T_k') \cap R(\beta,T_k')=\emptyset \\ C(\alpha,\beta), \text{ otherwise} \end{cases} \quad \text{식(3)}$$

이 방법은 L2 배열만을 가지고 처리하기 때문에 L2-tree 구성에 들어가기 전에 미리 배열에 계산한다. L3가 되지 않는 $\alpha\beta$에 관하여 $C(\alpha,\beta)$ 값이 -1이 되고 최소지지도를 만족하지 못해 L2로 처리되지 않으며 결국 L2-tree에 반영되지 않는다. 또한 L2의 출력은 최적화 전에 진행되기 때문에 L2의 출력은 최적화에 영향을 받지 않는다.

4. 실험

실험 환경은 펜티엄4 1.7Ghz에 1GB 메인 메모리이고 리눅스 환경에서 수행하였다. FP-growth와 L2-traverse, C3-traverse는 모두 C++로 구현되었으며 g++ 2.96 버전으로 컴파일 하였다. 본 논문에서 작성된 데이터는 [1]에서 소개된 데이터 생성 프로그램을 사용하여 생성하였다. 데이터의 특성은 다음과 같다. 사용된 데이터는 T15-I5-D100K이며 2,500개의 항목이 사용되었다. 트랜잭션의 평균길이 15, 잠재적인 빈발항목 길이 5, 트랜잭션의 개수 10만개임을 의미한다.

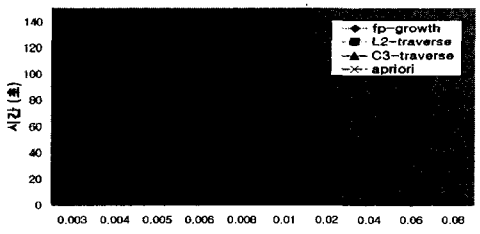

그림 2 T15.I5.D100K의 실행시간

그림 2에서 L2-traverse나 C3-traverse의 경우 FP-growth보다 배 이상 빠른 성능을 보임을 알 수 있으며 apriori의 경우 다른 알고리즘에 비해 실행 시간이 크게 증가한다. 지지도 4%에서 L1이 생성되며 지지도 1%에서 L2가 생성된다. 따라서 fp-growth는 지지도 4%에서, L2- traverse는 지지도 1%에서 실행시간이 급격히 증가한다. 그러나 C3-traverse의 경우 지지도 0.6%에서부터 L3가 가능한 L2가 시작됨을 알 수 있다. 결국 C3-traverse는 L2-traverse보다 효율적인 알고리즘임을 알 수 있다.

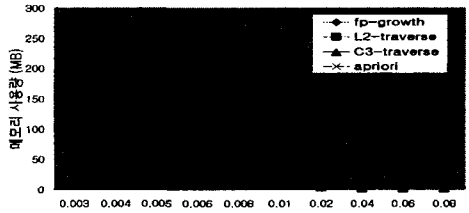

그림 3 T15.I5.D100K 메모리 사용량

그림 3은 메모리 사용량에 있어서도 FP-growth나 apriori보다 C3-traverse나 L2-traverse가 더 효율적임을 보여준다. 지지도 0.8%에서 C3-traverse와 L2-traverse 간에 메모리 차이가 미미한 반면 속도차이가 많이 발생하는 이유는 적은 개수의 L2가 발생하여 DB를 1회 더 스캔하면서 L2-tree 구성이 필요한 L2-traverse의 시간이 상대적으로 크게 나타나기 때문이다.

5. 결론

본 논문에서는 빈발항목집합 길이 2인 L2를 기반으로 하는 L2-tree를 제안하였다. 또한 L2-tree를 사용하여 빈발항목집합을 탐색하는 알고리즘인 L2-traverse를 제안하였다. L2-tree는 L2를 기반하기 때문에 L2가 적은 희소데이터 집합에서 적은 메모리 공간을 요구한다. 또한 L2-traverse 알고리즘은 FP-growth와 달리 별도의 추출 구조를 만들지 않고 단순히 L2-tree를 오직 한번의 깊이 우선 탐사 함으로써 빈발항목집합을 찾는다. L2-traverse에 대한 최적화 방법으로 C3-traverse 알고리즘을 제안하였다. 이 알고리즘은 L2-traverse와 거의 유사하나 L3가 되지 않을 L2를 L2-tree 구성 전에 미리 제거함으로써 메모리 공간을 10~40%, 속도는 10~50%정도의 향상되었다. 실험에서 L2-traverse 와 C3-traverse는 희소데이터 집합에서 FP-growth보다 매우 적은 양의 메모리를 필요로 하며 뿐만 아니라 시간적인 성능 향상을 보여주었다. 밀집데이터 집합에서의 실험은 FP-growth와 비슷하거나 다소 향상된 속도를 보여준다. 따라서 향후 밀집데이터 집합에서 보다 효율적으로 적용할 수 있도록 하는 알고리즘 최적화 방법이나 데이터의 특성에 따라 동적으로 알고리즘을 변경하는 기법에 대한 연구가 추가적으로 필요하다.

참고문헌

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proc. 1994 Int. Conf. Very Large DataBases (VLDB '94), pages 487-499. 1994.

[2] A. Savasere, E. Omiencinsky, and S. Navathe, An efficient algorithm for mining association rules in large databases, In Proceedings of the 21st VLDB Conference, pp. 432-444, 1995.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00), pages 1-12, 2000.

[4] R. Agawal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In J. of Parallel and Distributed Computing (Special Issue on High Perfomance Data Mining), 2000

[5] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases, Proc. 2001 Int. Conf. on Data Mining (ICDM'01), 2001.