희소 데이터 집합에서 효율적인 빈발 항목집합 탐사 기법

박 인 창[†] · 장 중 혁[††] · 이 원 석[†††]

요 약

빈발 항목집합 마이닝 분야의 주된 연구 주제는 수행과정에서의 메모리 사용량을 줄이고 짧은 수행 시간에 마이닝 결과 집합을 얻는 것으로서, 빈발항목 탐색을 위한 다수의 방법들은 Apriori 알고리즘에 기반을 둔 다중 탐색 방법들이다. 또한 최대 빈발 패턴의 길이가 길어질수록 마이닝 수행 시간이 급격히 증가되는 단점을 가진다. 이를 극복하기 위해서 이전의 연구에서 마이닝 수행 시간을 단축하기 위한 다양한 방법들이 제안되었다. 하지만, 다수의 이들 방법들은 희소 데이터 집합에서는 다소 비효율적인 성능을 나타낸다. 본 논문에서도 효율적인 빈발항목 탐색 방법을 제안하였다. 먼저 빈발항목 탐색을 위한 새로운 트리구조인 L_2-tree 구조를 제안하였으며, 더불어 L_2-tree를 이용하여 빈발 항목집합을 탐색하는 L_2-traverse 알고리즘을 제안하였다. L_2-tree 구조는 길이가 2인 빈발 항목집합 L_2에 기반하여 생성되는 것으로서 크기가 매우 작으며, 이를 활용한 L_2-traverse 알고리즘은 L_2-tree를 단순히 한번 탐색함으로써 전체 빈발 항목집합을 빠른 시간에 구한다. 또한 수행 시간을 보다 단축할 수 있는 방법으로 길이가 3인 빈발 항목집합 L_3가 될 수 없는 L_2 패턴들을 미리 제거하는 C_3-traverse 알고리즘도 제안하였다. 다양한 실험을 통해 제안된 방법들은 특히 L_2가 상대적으로 적은 희소 데이터 집합 환경일 때 기존의 다른 방법들보다 우수함을 검증하였다.

키워드 : 빈발 항목집합, 희소 데이터 집합, L_2-트리, L_2-탐색기법, C_3-탐색기법

Efficient Mining of Frequent Itemsets in a Sparse Data Set

In-Chang Park[†] · Joong-Hyuk Chang[††] · Won-Suk Lee[†††]

ABSTRACT

The main research problems in a mining frequent itemsets are reducing memory usage and processing time of the mining process, and most of the previous algorithms for finding frequent itemsets are based on an Apriori-property, and they are multi-scan algorithms. Moreover, their processing time are greatly increased as the length of a maximal frequent itemset. To overcome this drawback, another approaches had been actively proposed in previous researches to reduce the processing time. However, they are not efficient on a sparse data set. This paper proposed an efficient mining algorithm for finding frequent itemsets. A novel tree structure, called an L_2-tree, was proposed first, and an efficient mining algorithm of frequent itemsets using L_2-tree, called an L_2-traverse algorithm, was also proposed. An L_2-tree is constructed from L_2, i.e., a set of frequent itemsets of size 2, and an L_2-traverse algorithm can find its mining result in a short time by traversing the L_2-tree once. To reduce the processing more, this paper also proposed an optimized algorithm C_3-traverse, which removes previously an itemset in L_2 not to be a frequent itemsets of size 3. Through various experiments, it was verified that the proposed algorithms were efficient in a sparse data set.

Key Words : Frequent Itemset, Sparse Data Set, L_2-tree, L_2-traverse, C_3-traverse

1. 서 론

근래 들어 컴퓨터 및 관련 기기의 발달로 인해 데이터의 양은 급격히 증가하고 있다. 따라서 방대한 데이터에 내재된 지식을 효율적으로 획득하기 위해서는 자동화된 지식 획득 방법을 필요로 한다. 이러한 시대적 요구에 적절하게 대응할 수 있는 기술로 데이터 마이닝이 대두되고 있다. 데이터 마이닝은 데이터베이스와 인공지능, 통계적 기술을 이용하여 자동화된 지식 획득을 주된 목표로 하고 있다. 대표적인 데이터마이닝 알고리즘인 연관규칙[1-6]은 크게 두 가지의 처리단계를 거치게 된다. 하나는 빈발 항목집합(frequent itemset)을 찾는 것이고, 다른 하나는 빈발 항목집합으로부터 규칙을 생성하는 것이다. 빈발 항목집합이라고 하는 것은 일반적으로 지식탐색시스템의 사용자에 의해서 정의된 값인 최저 지지도(minimum support)를 만족하는 항목집합(itemset)을 말한다. 이러한 작업은 데이터베이스에서 항목의 출현 개수를 셈하는 작업이므로 데이터베이스 스캔과 함께 적지 않은 시간이 소모된다. 따라서 연관 규칙 탐사의 전체 성능은 사실상 빈발항목 탐색 단계를 얼마나 효과적으로 수행 할 수 있는가에 의해서 결정된다[7, 8].

† 정 회 원 : 삼성전자 선임연구원
†† 준 회 원 : 연세대학교 소프트웨어응용연구소 전문연구원
††† 종신회원 : 연세대학교 컴퓨터과학과 교수
논문접수 : 2005년 8월 24일, 심사완료 : 2005년 10월 24일

일반적으로 $I=\{i1, i2, \ , in\}$는 응용 분야에서 발생되는 단위 정보를 의미하는 항목(item)들의 집합을 나타내며, 트랜잭션 데이터베이스 $DB=\langle T1, T2, \ , Tk\rangle$는 항목들이 해당 응용분야에서 동시에 발생한 정보를 의미하는 트랜잭션들의 집합을 나타낸다. 이때 항목집합 (즉, 항목들의 집합이 패턴) A의 지지도는 전체 트랜잭션 데이터베이스 DB에서 해당 항목집합이 출현한 트랜잭션의 비율을 의미하며 $sup(A)$로 나타낸다. 하나의 항목집합 A는 해당 항목집합의 지지도 $sup(A)$가 사전에 정의된 지지도 임계값(즉, 최소지지도)보다 크거나 같을 때 빈발 항목집합이라 지칭한다. 빈발 항목집합 탐색 작업은 이와 같은 트랜잭션 데이터베이스 DB와 최소지지도가 주어졌을 때 해당 트랜잭션 데이터베이스에 포함된 모든 빈발 항목집합들을 구하는 작업으로 정의된다.

빈발 항목집합 탐사를 이전의 연구들 중 다수는 [9]에서 제안된 Apriori 속성에 기반을 두고 있다. 일반적인 apriori 속성 기반 알고리즘들은 빈발 항목집합의 최대 길이가 k일 때 $k+1$번만큼 반복적으로 스캔해야 하며 후보 항목집합생성과 갱신에 많은 비용이 든다. 이러한 단점을 개선하기 위해서 FP-growth[10] 방법 및 TreeProjection[11] 방법 등은 패턴-확장(pattern-growth) 방법을 사용하며, 후보항목을 생성하기보다는 발견된 빈발 항목집합에 기반하여 추출 데이터베이스를 만들고 패턴의 길이가 하나 늘어난 빈발 항목집합들을 찾는다. 이러한 방법들은 Apriori 기반 알고리즘들의 후보항목 생성 과정을 생략함으로써 마이닝 수행 과정에서의 비용을 줄인다. 하지만, FP-growth 방법에서는 길이가 1인 빈발항목에 기반하여 데이터 구조를 구성하므로 길이가 1인 항목이 대부분 빈발항목이 되는 경우 메모리 사용량이 크게 증가되며 빈발 항목집합 탐사에도 많은 시간이 소모되는 단점을 갖는다.

일반적으로 실제 응용 분야에서 발생되는 정보는 다수의 단위 정보들로 구성되는 복합 정보들은 거의 파악되지 않고 단위 정보 그 자체 혹은 적은 수의 단위 정보들로 구성되는 복합 정보들을 주로 포함한다. 이와 같이 분석 대상 데이터 집합이 짧은 길이의 유용한 지식을 주로 포함하며 길이가 긴 유용한 지식을 거의 포함하지 않는 경우를 희소 데이터 집합이라 한다. 일반적으로 이전에 제안된 빈발항목 탐사 방법들은 이러한 희소 데이터 집합에서는 효율적으로 적용되지 않는 단점을 지닌다.

본 논문에서는 빈발 항목집합탐사 방법에 대해서 길이가 2인 빈발 항목집합 L_2를 먼저 구하고, 이를 기반으로 모든 빈발 항목집합을 추출하는 방법을 제시한다. 일반적으로 희소 데이터 집합에서 길이가 2인 빈발 항목집합 L_2의 개수는 상대적으로 길이가 1인 빈발항목 L_1보다 매우 적으며, L_1과 L_2는 메모리 상에서 빠르게 탐사 할 수 있다. 따라서 본 논문에서 제안되는 방법은 분석 대상 데이터 집합에 대한 탐색 횟수는 기존의 빈발 항목집합 탐사 알고리즘인 FP-growth 알고리즘에 비해 증가되지만 마이닝 수행 시간을 감소시킬 수 있다. 또한 L_3가 될 수 없는 L_2를 고려대상에서 먼저 제거하는 최적화 방법인 C_3-traverse 알고리즘도 함께 제안하며, 다양한 특성을 갖는 데이터 집합에 대한 실험을 통해 본 논문에서 제안하는 C_3-traverse 알고리즘이 희소 데이터 환경에서뿐만 아니라 밀집 데이터 환경에서도 효율적임을 검증한다.

본 논문의 구성은 다음과 같다. 제 2절에서는 빈발 항목집합 및 연관규칙 탐사와 관련된 기존의 연구들을 소개하고 문제점을 설명하며, 제 3절에서는 길이가 2인 빈발 항목집합 기반의 데이터 구조인 L_2-tree의 구성 방법을 제시한다. 제 4절에서는 L_2-tree를 이용한 빈발 항목집합 탐사 알고리즘인 L_2-traverse 알고리즘과 이를 최적화한 C_3-traverse 알고리즘을 제안한다. 제 5절에서는 제안된 알고리즘에 대한 실험결과를 제시하여 알고리즘의 성능을 평가하며, 끝으로 제 6절에서는 결론과 함께 향후 연구 방향에 대하여 기술한다.

2. 관련 연구

연관규칙 탐사와 관련하여 대표적인 알고리즘들은 Apriori[9] 속성에 기반한 방법들로서 Apriori 알고리즘, ArioriTID[9] 방법 및 AprioriHybrid[9] 방법 등이 있다. Apriori 속성에 기반한 보다 향상된 방법으로 분할(Partition) 알고리즘[12] 및 DIC(Dynamic Itemset Counting) 알고리즘[13]이 제안되었다. 이들 방법들의 전체 데이터베이스 탐색 회수를 살펴보면 최대 빈발 항목집합의 길이가 m일 때 Apriori는 $m+1$번, 분할 알고리즘은 2회, DIC는 Apriori보다 적은 회수를 가진다. 하지만 이러한 Apriori 기반 방법들의 공통된 단점은 미리 많은 수의 후보 항목집합들을 생성해야하고 각 후보 항목에 대해 빈발 항목 여부 판단해야 한다. 따라서 Apriori 기반 방법들의 주된 비용은 데이터 스캔 회수보다는 많은 후보 항목집합을 생성하는데 있으며, 후보 항목집합 생성은 방대한 크기의 메모리 공간을 필요하므로, 후보 항목에 대한 검색 및 갱신에 많은 시간이 소모된다.

Apriori 속성에 기반하지 않는 방법으로서 TreeProjection[11] 방법이 제안되었다. 해당 방법은 먼저 항목명에 대한 오름차순으로 구성된 트리를 구성하고 트리를 탐사하면서, 각 노드를 만족하는 트랜잭션들을 데이터베이스로부터 추출하여 해당 노드의 부분 데이터베이스로 구축한다. 길이가 k인 항목집합에 해당하는 노드에서 자신의 빈발 항목집합을 찾기 위해 2차원 매트릭스를 구성하여 지지도를 계산하며, $k+1$ 빈발 항목집합을 생성하고 각각에 해당하는 자식 노드를 생성하여 트리를 계속 확장해 나간다. 이러한 Tree Proejction의 단점은 첫째로 데이터베이스가 매우 크거나 빈발 항목집합이 많을 때 여전히 많은 계산량을 필요로 하며, 둘째로 TreeProjection에서는 한 트랜잭션이 많은 빈발 항목집합을 가질 때 트리의 많은 노드에서 중복하여 추출 되는 단점을 갖는다[10].

근래에 발표된 FP-growth[10]는 기존의 Apriori 기반 방법과 TreeProjection보다 여러 면에서 효율적인 알고리즘이며 주된 이유는 모든 후보항목을 생성하지 않는데 있다. 일

반적으로 효율적인 빈발 항목집합 탐사를 위해서는 후보 항목집합에 대한 생성과 관리에 적합한 데이터 구조가 필요하다. 기존의 알고리즘 중에서 후보 항목집합을 표현하기 위한 구조로 많이 사용되는 격자(lattice)는 방향성 비 순회 그래프(DAG)이며 인접 격자(adjacency lattice)로 불리기도 한다. 격자는 빈발여부를 확인해야할 모든 후보 항목집합을 노드로 표현한 구조이므로 후보 항목집합이 많아 질 경우 탐색공간이 매우 커지는 것이 단점이 있다[10, 11, 14]. 또한, 길이가 k인 후보 항목집합은 해당 후보 항목집합의 모든 부분후보 항목집합을 메모리 상에 유지해야하기 때문에 많은 메모리 공간을 필요로 한다. 즉 격자에서는 빈발 항목집합의 크기가 길어질수록 지수적으로 시간과 공간이 증가하게 되며 이러한 단점을 제거한 구조는 FP-growth에서 사용된 FP-tree[10]이다. FP-tree는 각 트랜잭션별로 길이가 1인 빈발항목만을 뽑아 지지도 내림차순으로 정렬한 순서대로 트리의 경로를 생성한다.

FP-tree의 장점은 각 트랜잭션을 구성하고 있는 항목들이 유사할 경우 트리 상에서 공유되는 부분이 많아지고 FP-tree의 메모리 요구량은 적어진다. 또한 FP-tree를 구성하는데 단지 2회의 데이터베이스 탐사가 필요하며 추가적인 데이터베이스 탐사를 요구하지 않는다. FP-tree의 단점은 트랜잭션간에 항목의 공유가 적어지게 될수록 FP-tree의 메모리 요구량이 트랜잭션 데이터베이스의 크기에 근접하게 된다. 일반적으로 희소 데이터 집합에서는 밀집 데이터 집합보다 항목의 공유가 이루어질 확률이 적어지게 되며, 결국 희소 데이터 집합에서 FP-tree는 방대한 메모리 공간을 필요로 하게 된다. 또한, 빈발 항목집합을 탐색하는 과정에서 조건 패턴(conditional pattern)을 기반으로 FP-tree의 부분트리 형태인 조건 FP-tree는 FP-tree로부터 물리적으로 추출되며 조건 패턴이 빈발패턴이 되어 다른 후보패턴으로 확장할 때, 이미 추출된 조건 FP-tree에서 관련된 경로가 다시 추출되는 재귀적 과정을 거쳐야 한다. 이러한 작업에서 FP-growth는 많은 시간을 소모하게 되며 최대 빈발 항목집합의 길이가 길수록 FP-tree의 경로와 노드가 많아지고 조건 FP-tree의 크기도 매우 커지며 더 많은 회수의 재귀적 호출이 수행됨으로 빈발항목탐사 시간이 크게 늘어나게 된다. 그러므로 FP-growth 방법에서는 희소 데이터 집합 환경에서 방대한 메모리가 요구되고 처리 시간 또한 크게 증가된다[14].

3. L_2-tree의 구조 및 생성

길이가 n인 빈발항목들의 집합을 L_n으로 나타내고 길이가 n인 후보항목들의 집합을 C_n으로 나타낸다. [10]에서 제안된 FP-tree는 길이가 1인 빈발 항목 집합 L_1을 기반으로 단위항목들의 지지도를 고려하여 구성된 트리 구조로서 해당 트리에서는 L_1에 포함되는 단위항목들 중에서 L_2를 형성하지 않는 항목들도 트리구성에 참여한다. 하지만, L_2를 형성하지 않는 단위항목들은 이후 보다 긴 빈발항목을 구성하는 과정에서도 불필요한 단위항목으로서 이들을 제외하고 지지도기반 트리를 구성하는 경우 보다 축약한 트리를 구성할 수 있으며, 따라서 지지도 기반 트리 구조를 이용한 빈발항목 탐색 시간도 단축된다.

본 논문에서는 L_2를 형성하는 단위항목들로 구성되는 지지도 기반 트리 구조인 L_2-tree를 제안한다. 빈발항목 탐색을 위한 하나의 트랜잭션 데이터베이스에 대해서 L_2-tree 생성 과정은 간단히 다음과 같다.

1. 해당 트랜잭션 데이터베이스를 한번 탐색하여 L_1을 탐색한다.
2. L_1을 기반으로 길이가 2인 후보항목 집합 C_2를 생성하고 해당 트랜잭션 데이터베이스를 탐색하여 L_2를 구한다. 이때, L_1에 기반한 매트릭스 구조를 이용하여 후보항목 집합 C_2를 생성함으로써 후보항목 생성에 따른 자원소모를 최소화한다. 본 논문에서는 이러한 매트릭스 구조를 C_2-matrix라 한다.
3. 트랜잭션 데이터베이스를 탐색하여 각 트랜잭션을 L_2 형성에 참여하는 단위항목들을 추출하고 해당 단위항목들을 지지도 내림차순으로 재정렬함으로써 L_2기반 트랜잭션으로 변환하며 변환된 트랜잭션들을 이용하여 지지도 기반 트리 구조인 L_2-tree를 구성한다.

3.1절에서는 C_2-matrix 및 L_2 기반 변환 트랜잭션에 대해서 기술하며 3.2절에서는 변환 트랜잭션을 이용한 지지도 기반 트리인 L_2-tree에 구조 및 생성 과정에 대해서 기술한다. 끝으로 3.3절에서 예제를 통해 C_2-matrix 및 L_2-tree 생성 과정을 살펴본다.

3.1 C_2-matrix 및 L_2 기반 변형 트랜잭션

트랜잭션 데이터베이스를 한번 탐색하여 구해진 L_1을 기반으로 L_2 탐색을 위한 후보항목 집합 C_2를 구하는 과정은 일반적으로 트리 형태의 격자 구조를 이용한다. 본 논문에서는 후보항목 집합 C_2를 L_1으로 구성되는 2차원 매트릭스 형태로 표현한다. 트리 형태의 격자로 표현된 C_2를 이용하여 트랜잭션 데이터베이스를 탐색하여 각 후보항목의 출현빈도 수를 분석하는 경우 트리 탐색을 위한 시간 낭비가 따른다. 하지만, 매트릭스 형태로 표현된 C_2를 이용하여 트랜잭션 데이터베이스를 탐색하는 경우는 별도의 트리 탐색 작업을 필요로 하지 않으므로 L_2 탐색 시간을 줄일 수 있다. 하나의 트랜잭션 데이터베이스 DB에 대한 길이 1인 빈발 항목 집합 L_1에 대해서 $|L_1|$이 해당 빈발항목들의 집합에 포함되는 빈발항목의 수를 나타낼 때 C_2-matrix는 [정의 1]에서와 같이 정의된다.

[정의 1] (C_2-matrix) C_2-matrix는 $(|L_1|-1)\times(|L_1-1$

크기의 상삼각행렬(Upper triangular matrix)로서 지지도 내림차순으로 정렬된 빈발항목리스트 $L_1=[a_1, a_2, \ldots, a_{|L_1|}]$, $sup(a_i)\geq sup(a_{i+1})$에 대해서, C_2-matrix $U[a_1..a_{|L_1|-1}, a_2..a_{|L_1|}]$은 다음과 같이 정의된다.

$$U[a_i, a_j]=\begin{cases} sup(a_i a_j) & \text{if } sup(a_i)>sup(a_j) \\ 0 & otherwise \end{cases}$$ □

FP-growth의 경우 L_2가 존재하지 않더라도 L_1이 존재하면 FP-tree를 구성하게 되지만 L_2가 공집합인 경우에는 크기가 3 이상인 빈발 항목집합도 존재하지 않으므로 빈발항목집합 탐사를 위한 데이터 구조인 L_2-tree를 구성할 필요가 없다. 따라서 C_2-matrix를 통해서 L_2가 공집합이 아님을 확인한 후 트리를 구성한다. 한편, 길이가 2인 빈발 항목집합을 구성하는 항목을 **L_2-enabling 항목**이라고 정의하고 L_2-enabling 항목들의 집합을 **L_2-enabling 항목 집합** L_2E라고 정의한다. 만약 L_2가 존재한다면 각 트랜잭션에서 L_2-enabling 항목이 아닌 항목들을 제거하고 [정의 2]에서와 같은 L_2기반 트랜잭션(L_2-transaction)으로 각 트랜잭션을 재구성한다. 또한, 각 L_2-transaction에서 단위항목의 위치에 따라 선행항목, 후행항목, 후행항목 집합을 [정의 3]에서와 같이 정의한다. C_2-matrix를 통해서 각 선행항목에 대한 후행항목과 후행항목 집합을 효과적으로 파악할 수 있으며 이러한 정보를 기반으로 데이터베이스의 각 트랜잭션 별로 L_2-tree 구조에 입력될 L_2 기반 정보인 후행항목 리스트와 후행항목 리스트 집합을 [정의 4] 및 [정의 5]에서와 같이 생성한다.

[정의 2] (L_2 기반 변형 트랜잭션) 트랜잭션 $T_k\in DB$의 항목들 중에 L_2-enabling 항목들만을 모아 지지도 내림차순으로 정렬한 트랜잭션 $\overline{T_k}$를 T_k에 대한 L_2 기반 변형 트랜잭션 이라 정의한다.

$$\overline{T_k}=[x_i \mid x_i\in T_k \text{ and } x_i\in L_2E \text{ and } sup(x_i)\geq sup(x_{i+1})]$$ □

[정의 3] (선행항목, 후행항목, 후행항목 집합) 트랜잭션 $T_k\in DB$의 변형 트랜잭션 $\overline{T_k}$ 에 발생한 길이 2인 항목집합 $a, b\in L_2$에 대해서 $sup(a)\geq sup(b)$를 만족할 때 a를 b의 ***선행항목***이라 정의하고 b를 a의 ***후행항목***이라 정의한다. L_2 및 $\overline{T_k}$가 주어졌을 때, 하나의 선행항목 $a\in\overline{T_k}$의 모든 후행항목들을 $\overline{T_k}$에 대한 a의 ***후행항목 집합*** $\gamma_a(\overline{T_k})$이라 정의한다.

$$\gamma_a(\overline{T_k})=\{x \mid x\in\overline{T_k},\ ax\in L_2 \text{ and } sup(a)\geq sup(x)\}$$ □

[정의 4] (변형 트랜잭션 $\overline{T_k}$에서 항목 a의 후행항목 리스트) 변형 트랜잭션 $\overline{T_k}$의 한 항목 $a\in\overline{T_k}$에 대한 후행항목 리스트 $tail-list(\overline{T_k} \mid a)$는 해당 항목 a 및 a의 모든 후행항목들을 지지도 내림차순으로 정렬한 리스트로 정의된다. 단, 앞서 기술한 바와 같이 L_2-tree를 이용한 접근법은 길이가 3이상인 빈발항목을 구하기 위한 것이며, 따라서 후행항목 집합의 크기가 2보다 작은 경우는 해당 빈발항목이 존재하지 않는 경우로서 후행항목 리스트가 정의되지 않는 것으로 간주한다.

$$tail-list(\overline{T_k}|a)=[(a, x_i) \mid x_i\in\gamma_a(\overline{T_k}),\ sup(x_i)\geq sup(x_{i+1}), 1\leq i\leq|\gamma_a(\overline{T_k})|]$$ □

[정의 5] (변형 트랜잭션 $\overline{T_k}$에 대한 후행항목 리스트 집합) 변형 트랜잭션 $\overline{T_k}$에 대한 후행항목 리스트들의 집합 $Tail-list-set(\overline{T_k})$은 해당 변형 트랜잭션에서 나타나는 모든 후행항목 리스트들의 모음으로서 다음과 같이 정의된다.

$$Tail-list-set(\overline{T_k})=\bigcup_{i=1}^{m-2} tail-list(\overline{T_k} \mid a_i),\ where\ m=|\overline{T_k}|$$ □

예를 들어 항목집합 $I=\{a, b, c, d, e, f, g\}$, $L_2=\{cb, cd, ca, bd, bf, da, fg\}$일 때 트랜잭션 $T_k=\{a, b, c, d, e, f\}$의 변형 트랜잭션이 $\overline{T_k}=[c, b, d, a, f]$라고 가정하면 $\overline{T_k}$에서 항목 c의 후행항목 집합 $\gamma_c(\overline{T_k})$은 주어진 L_2에서 항목 c를 포함하는 항목집합이 $\{cb, cd, ca\}$이고 b, d, a가 모두 $\overline{T_k}$에 포함되므로 [정의 3]에 의해서 $\gamma_c(\overline{T_k})=\{b, d, a\}$가 된다. 따라서 이러한 방법으로 $\overline{T_k}$에 대한 각 선행항목에 대한 후행항목 집합은 $\gamma_c(\overline{T_k})=\{b, d, a\}$, $\gamma_b(\overline{T_k})=\{d, f\}$, $\gamma_d(\overline{T_k})=\{a\}$, $\gamma_a(\overline{T_k})=\{\}$, $\gamma_f(\overline{T_k})=\{\}$가 된다. [정의 4]에 의해서 $tail-list(\overline{T_k} \mid c)=[c, b, d, a]$가 되고 $tail-list(\overline{T_k} \mid b)=[b, d, f]$가 된다. 선행항목이 d, a 그리고 f인 경우 후행항목 집합의 크기가 2보다 작으므로 [정의 4]에 의해서 생성되지 않고 *null*이 된다. 또한 [정의 5]에 따라 $\overline{T_k}$에 대한 후행항목 리스트 집합은 $Tail-list-set(\overline{T_k})=\{[c, b, d, a], [b, d, f]\}$가 된다.

3.2 L_2-tree

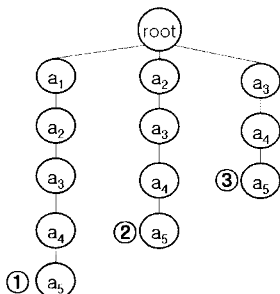
L_2-tree는 각 트랜잭션에 대한 변형 트랜잭션에서 구한 후행항목 리스트 집합의 원소 리스트들을 하나의 경로로 표현하는 구조이다. 리스트내의 각 항목은 하나의 노드가 되며 해당되는 항목명과 지지도를 가지게 된다. 각 노드의 지지도는 L_2-tree의 루트 노드로부터 해당 노드까지 표현된

항목집합의 지지도를 갖는다. FP-tree 구조에서는 하나의 트랜잭션이 하나의 경로로 표현되는 반면에, L_2-tree에서는 각 변형 트랜잭션의 후행항목 리스트 집합에 있는 각 리스트들을 각각 별도의 경로로 나타내며 L_3이상의 빈발 항목 집합만을 탐사하게 된다. L_2-tree는 모든 후보 항목집합을 표현하지 않고 L_2를 만족하는 변형 트랜잭션들의 정보만을 표현하므로 L_2-tree에서 각 항목집합의 정확한 지지도를 구할 수 있는 방법을 제시해야 한다. 먼저 후보 항목집합과 L_2-tree의 경로와의 관계를 아래와 같이 정의한다.

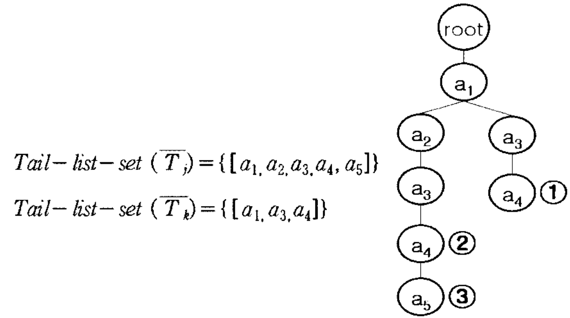
[정의 6] (후보 항목집합 A에 대한 L_2-tree상에서의 경로 $path(A)$) L_2-tree에서 항목집합 A에 대한 ***항목집합경로*** $path(A)$는 항목집합 A에서 가장 높은 지지도를 갖는 항목의 노드로부터 시작하여 가장 낮은 지지도를 갖는 항목의 노드에 이르는 의미한다. 이들 경로에 존재하는 항목들(즉, 항목집합 A를 구성하는 항목들)은 지지도 내림차순의 순서가 유지된다. □

(그림 1)은 변형 트랜잭션 $\overline{T_k}$에 대한 후행항목 리스트 집합 $Tail-list-set\ (\overline{T_k})$의 각 리스트를 트리 형태로 구조화한 그림이다. 트리의 항목 개수는 각각 5개, 4개, 3개이고 중복되는 경로가 없으므로 루트 노드를 포함해 모두 13개의 노드가 생기며 하나의 트랜잭션에 나타났으므로 각 노드의 지지도는 모두 1이 된다. 트리상에서 항목집합 $a_3a_4a_5$의 항목을 포함하는 루트로부터의 경로는 ①, ②, ③이지만 [정의 6]에 의하여 항목집합 $a_3a_4a_5$의 $path(a_3a_4a_5)$는 경로 ③이 된다. 경로 ①의 의미는 데이터베이스에 항목집합 $a_3a_4a_5$의 선행항목으로 a_1과 a_2가 출현했음을 나타내며 경로 ②는 $a_3a_4a_5$의 선행항목으로 항목 a_2가 출현했을 경우를 나타낸다. 마찬가지로 경로 ③은 $a_3a_4a_5$가 선행항목 없이 출현한 경우를 나타낸다. 따라서 ①의 출현은 ②와 ③을 동시에 만족하고 ②의 출현은 ③을 만족하므로 항목집합의 출현 횟수를 구할 때 해당 항목집합에 대한 경로만을 고

$\overline{T_k} = [\ a_1,\ a_2,\ a_3,\ a_4,\ a_5\ ]$
$Tail-list-set\ (\overline{T_k}) = \{[a_1, a_2, \underline{a_3, a_4, a_5}], [a_2, \underline{a_3, a_4, a_5}], [\underline{a_3, a_4, a_5}]\}$



(그림 1) 단일 트랜잭션에서의 L_2-tree 구조



(그림 2) 다수 트랜잭션 집합에서의 L_2-tree 구조

려하게 된다. 다시 말하면 지지도를 찾고자 하는 항목집합의 선행항목으로부터 시작되는 항목집합경로로 제한했을 때 정확한 지지도를 구할 수 있다.

(그림 2)는 두 개의 변형 트랜잭션 $\overline{T_j}$과 $\overline{T_k}$에 대한 후행항목 리스트 집합이 각각 주어졌을 때 항목집합 $a_1a_3a_4$의 지지도를 구하는 예이다. $\overline{T_j}$와 $\overline{T_k}$의 후행항목 리스트 집합을 트리 형태로 구성하면 두 리스트간 중복되는 선행경로 (a_1)은 한 개의 노드로 생성되어 총 8개의 노드가 생성된다. (그림 2)의 트리에서 [정의 6]을 만족하는 항목집합 $a_1a_3a_4$의 항목집합경로는 ①과 ②가 된다. 경로 ②가 의미하는 것은 $a_1a_2a_3a_4$가 출현한 것이고, ③이 의미하는 것은 $a_1a_2a_3a_4a_5$가 출현하였음을 나타내므로 이들이 동일한 트랜잭션에서 나타났다면 경로 ③을 만족하면서 경로 ②를 만족하게 되어 출현빈도가 중복되어 증가하며, 이러한 문제를 해결하기 위해서 지지도를 구하고자 하는 항목집합 A의 항목집합경로 $path(A)$만을 고려해야 한다. 따라서, 항목집합 A의 항목집합경로 $path(A)$는 항목집합 A의 지지도 내림차순 항목리스트를 p라고 정의하고, A의 항목집합경로 $path(A)$에 나타나는 항목리스트를 q로 표현할 때 다음의 세 가지 조건을 동시에 만족해야 한다. $first(r)$과 $last(r)$은 항목리스트 r의 첫 번째 항목과 마지막 항목을 나타낸다.

i) $p \subseteq q$, ii) $first(p) = fisrt(q)$, iii) $last(p) = last(q)$

L_2-tree에서 항목집합 A의 항목집합경로 $path(A)$들 중에서 A의 항목들만으로 구성된 경로를 자기 경로라 정의하고 A의 항목들 이외의 항목을 포함하는 경로를 분산경로라 정의한다. 이때 자기 경로상의 마지막노드의 지지도를 A의 자기지지도 $selfsup(A)$라 정의하고 항목집합 A의 모든 분산경로의 마지막노드들의 지지도 합을 분산지지도 $distsup(A)$라 정의한다. 따라서 항목집합 A의 지지도 $sup(A)$는 다음과 같이 정의된다.

$$sup(A) = selfsup(A) + distsup(A)$$

L_2-tree는 특정 항목집합의 분산지지도를 효과적으로 찾기 위해 L_2-tree상에 서로 연관된 노드들을 연결한 구조를

별도로 갖는다. 지지도를 구하고자 하는 대상 항목집합의 첫 번째 항목이 동일한 경로들만을 순회하기 위해 L_2-tree의 깊이가 1인 각 노드를 루트로 하는 부분트리당 하나의 독립적인 ***동일항목연결 구조***를 만든다. 그러므로 L_2-tree 깊이 1의 노드 수만큼 별도의 동일항목연결 구조들이 구성된다. 각 동일항목연결 구조는 지지도를 찾고자 하는 항목 집합의 길이가 지지도 계산에 연관된 경로의 깊이보다 항상 작아야 하므로 연관된 경로들을 효과적으로 탐색하기 위해 각 동일항목연결 구조는 각 항목별로 L_2-tree상에 있는 동일한 항목노드들을 깊이에 대해서 내림차순으로 연결한 리스트 구조이다. 이러한 동일 항목연결 구조를 효과적으로 탐색하기 위해 깊이 1의 각 노드를 루트로 하는 부분트리별로 *시작항목이름, 연결될 항목이름*과 *동일 항목연결 구조 시작 포인터*를 저장하는 ***동일 항목연결 구조 헤더 테이블***을 유지한다. 이때 동일항목 연결구조에 대한 시작 포인터는 트리상의 동일항목노드들 중에서 가장 깊은 깊이의 노드를 가르키며 특정 항목집합에 해당하는 경로들을 찾을 때에 대상 항목집합의 끝 항목이 가장 깊게 나타난 경로부터 탐색하여 모든 분산경로를 찾을 수 있도록 한다. 따라서 항목집합의 크기보다 작은 길이의 경로를 탐색하지 않으므로 불필요한 검색 공간을 줄일 수 있는 이점이 있다.

모든 후보 항목집합을 생성하고 관리하는 격자와는 달리 변형 트랜잭션들의 정보로 구성된 L_2-tree에서는 지지도를 구하고자 하는 후보 항목집합에 해당하는 자기경로가 트리에 실제로 존재하지 않을 수 있다. 이러한 경우 후보 항목집합에 대한 분산경로들을 찾아 분산지지도를 파악하여 빈발 여부를 판별한다. 즉, 조건 패턴 기반의 *조건 FP-tree*를 메모리에 동적으로 각각 생성하여 빈발항목을 탐색하는 FP-tree를 이용한 빈발항목 탐색에서와는 달리 L_2-tree를 이용하는 경우 분산경로에 기반하여 후보 항목집합의 지지도를 구함으로써 추가적인 구조 없이 한번의 tree 순회만으로 해당 항목집합의 지지도를 구할 수 있다.

위와 같은 특성을 가지는 빈발 항목집합 탐사를 위한 데이터 구조 L_2-tree는 다음과 같이 정의된다.

i) L_2-tree는 하나의 루트 노드를 갖는다.

ii) 루트 노드로부터 시작되는 각 경로는 변형 트랜잭션의 후행리스트집합 $Tail-list-set(\overline{T_k})$에 따라 생성되며, 각 리스트간 동일한 선행 경로는 하나의 경로로 공유되어 구성된다.

iii) L_2-tree의 깊이가 3 이상인 각 노드는 항목명, 출현 횟수, 동일 항목연결 포인터를 갖는다. 항목명은 노드에 해당하는 항목 이름이고 출현 횟수는 깊이 1의 항목으로 시작하여 자신이 있는 노드까지의 경로에 해당하는 항목집합이 데이터베이스의 트랜잭션에 실제로 출현한 회수를 나타낸다. 동일 항목연결 포인터는 동일항목연결 구조에 의해 다른 경로에 있는 동일 항목에 대한 다음 노드를 가르키는 포인터이다.

iv) L_2-tree의 깊이 1의 각 항목 노드 a마다 자신을 루트로 하는 부분트리에 대하여 동일 항목연결 구조 헤더 테이블 HT(a)를 가지며 자신의 항목 이름 a, 동일항목 연결 구조, 항목 이름 리스트 및 이들 각각에 대한 동일 항목연결 시작 포인터들을 갖는다.

(그림 3)은 L_2-tree 구성 알고리즘에 대한 의사 코드(pseudo code)이다.

```
입력 : 트랜잭션 데이터베이스 DB와 최소지지도 ξ, C2-matrix U
출력 : L2-tree

Procedure gen-L2-tree ( C2-matrix U, ξ)
begin
    create the root node of an L2-tree
    For each Tk ∈ DB
        transform Tk to T̄k and find Tail-list-set(T̄k)
        using C2-matrix U
        For each list l in Tail-list-set(T̄k)
            currentNode := root
            currentItem := first item of list l
            For each item i in l
                if currentItem ∈ children of currentNode then
                    follow and update the count of currentItem
                    node in the subpath
                else
                    make a new node and set the count of each
                    node to 1
                endif
                currendNode := node of currentItem
                currentItem := next item in l
            end
        end
    end
    return root node
end
```

(그림 3) L_2-tree 생성 알고리즘

3.3 C_2-matrix 및 L_2-tree 생성 예제

<표 1>의 트랜잭션 데이터베이스 DB에 대해서 $\xi=2$일 때 먼저 첫 번째 데이터베이스 스캔을 통해 빈발항목 c, e, a, b, d, g 각각의 출연빈도 수 4, 4, 3, 3, 3, 3을 구하고 이들 빈발항목들로 (그림 4)의 C_2-matrix를 구성한 후 L_2를 탐사한다. C_2-matrix의 행은 각 변형 트랜잭션에서 선행항목, 열은 선행항목에 대한 후행항목을 나타낸다. 예를 들어 선행항목 c와 후행항목 a가 만나는 요소값은 1이며, 이것은 항목집합 ca의 지지도가 된다. 따라서 주어진 DB에서 만족된 L_2는 ce, cb, cd, cg, ea, ed, bg이며, L_2-enabling 항목 집합은 $L_2E=\{a,b,c,d,e,g\}$가 된다.

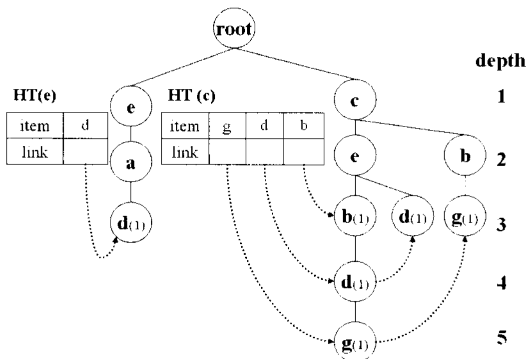
L_2가 구해진 후 각 트랜잭션을 재탐색하여 후행리스트 집합 $Tail-list-set$을 다음과 같이 구한다. 먼저 T_{100}에서 선행항목 e의 후행항목은 (그림 4)의 C_2-matrix를 참고하면 선행항목 e에서 최소지지도 2를 만족하는 항목은 a와 d이므로 [정의 4]에 의해서 $\gamma_e(\overline{T_{100}})=(a,d)$가 되며 $Tail-list-set(\overline{T_{100}})=\{[e,a,d]\}$이 된다. $[e,a,d]$는 루트노드

〈표 1〉 트랜잭션 데이터베이스 *DB*

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

선 \ 후	e	a	b	d	g
c	3	1	2	2	2
e		2	1	3	1
a			1	1	1
b				1	2
d					1

(그림 4) C_2-matrix



(그림 5) 〈표 1〉의 데이터에 대한 L_2-tree

부터 각각의 항목을 순서대로 L_2-tree에 경로를 생성하며 먼저 깊이 1인 노드 *e*에 대한 동일 항목 연결 헤더 테이블이 존재하지 않으므로 노드 *e*에 대한 헤더 테이블 HT(e)를 생성한다. $[e,a,d]$의 경로를 만드는 과정은 먼저 루트 노드의 자식 노드로 노드 *e*를 만들고 *e*의 자식노드로 새로운 노드 *a*를 만들지만 e와 a가 깊이 3 미만의 노드임으로 지지도는 파악하지 않는다. 같은 방법으로 노드 *d*를 *a*의 자식노드로 생성하며 *d*는 깊이 3 이상인 노드이므로 동일 항목 연결 구조로 연결하고 지지도를 1로 설정한다. T_{200}에 대해서 $\gamma_c(\overline{T_{200}})=(e)$가 되며 [정의 4]에 의해서 후행항목 리스트가 생성되지 않고 T_{300}의 리스트 역시 생성되지 않는다. T_{400}에서의 $Tail-list-set\ (\overline{T_{400}})=\{[c,e,b,d,g]\}$가 된다. 이때 $[e,b,d,g]$가 후행항목 리스트가 되지 않는 이유는 C_2-matrix를 참조하여 $\gamma_e(\overline{T_{400}})=(d)$임을 알 수 있으므로 항목이 2개 미만이 되어 후행항목 리스트가 되지 못하며 $[b,d,g]$도 $\gamma_b(\overline{T_{400}})=(g)$가 됨으로 동일한 이유로 후행항목 리스트가 되지 못한다. 따라서 $[c,e,b,d,g]$를 루트 노드로부터 해당 항목들에 대한 노드들을 순서대로 만들고 HT(c)를 생성하며 깊이 3 이상의 노드인 *b, d, g*에 대해 HT(c)의 각 동일 항목 연결 시작 포인터를 만든다. T_{500}은 $Tail-list-set\ (\overline{T_{500}})=\{[c,e,d]\}$이므로 $[c,e,d]$ 경로가 입력된다. 이때 루트로부터 경로 [*c, e*]가 이미 존재하므로 노드 *d*만을 노드 *e*의 자식노드로 새롭게 생성하고 깊이가 3 이상이므로 HT(c)의 *d*항목에 대한 동일항목 연결구조에 깊이내림차순으로 추가하고 지지도를 1로 설정한다. T_{600}에서 대해서도 $\{c,b,g\}$ 항목집합에 해당하는 경로가 앞서 기술한 것과 유사하게 처리된다. 이러한 과정을 통해 (그림 5)와 같은 L_2-tree가 생성된다.

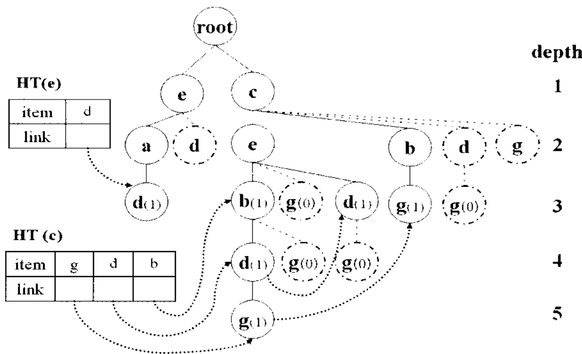
4. L_2-tree를 이용한 빈발 항목집합 탐사 방법

본 절에서는 L_2-tree에서 모든 빈발 항목집합을 탐사하는 L_2-traverse 알고리즘은 소개하고 이를 최적화한 C_3-traverse 방법을 제안한다. 이들 방법들은 L_2-tree에 대한 깊이 우선 탐사로 모든 빈발 항목집합을 탐사하며, L_2-tree는 L_2의 수가 적은 데이터 집합에서 매우 효율적으로 메모리 공간 및 수행시간을 감소시킨다.

4.1 L_2-traverse 알고리즘

L_2-tree의 깊이 1과 2에 해당하는 L_1과 L_2의 지지도는 미리 1차원 배열과 2차원 C_2-matrix에서 파악된 상태임으로 L_2-tree에서 이들 노드들의 지지도를 별도로 구하지 않아도 된다. L_3이상의 빈발 항목집합들은 L_2-tree를 순회하여 찾으며 L_2-tree는 L_1에 속한 항목들의 지지도 내림차순 기반으로 깊이 우선 탐색을 통해 파악한다. C_3 이상의 후보 항목집합의 정확한 지지도는 3절에서 설명하였듯이 대상 후보 항목집합의 자기지지도와 분산지지도의 합으로 구해진다. 빈발 여부가 판단되어져야 하는 후보 항목집합에 해당하는 경로가 L_2-tree상에 존재하지 않으면 분산지지도만을 구하며 지지도로 계산한다. 만약 현재노드에 해당되는 항목집합이 빈발 항목집합을 구성하지 않음을 확인한 후에는 그 노드의 모든 자식노드들도 빈발 항목집합이 되지 않기 때문에 자식 노드들을 탐색할 필요가 없다. 또한, L_2-matrix의 현재 노드와 자식 노드로 구성되는 2-항목집합(2-itemset)의 지지도가 최소지지도보다 낮을 경우에도 루트로부터 자식 노드까지의 항목집합이 빈발 항목집합이 되지 못하므로 더 이상 해당 경로를 탐색하지 않는다.

(그림 6)은 (그림 5)의 L_2-tree에서 L_2-traverse방법으로 빈발 항목집합을 탐색하는 예를 보여준다. 먼저 루트 노드를 방문하고 루트 노드의 자식 노드로써 L_1에서 가장 높은 지지도를 가지는 항목인 (*c*)로 이동한다. (*c*)로 이동한 후에는 (그림 4)의 C_2-matrix를 참고하여 가장 높은 지지도



(그림 6) 빈발 항목집합 탐사를 위한 L_2-traverse 알고리즘

를 갖는 *(ce)*로 이동한 후 C_2-matrix를 통해 노드 *(ce)* 이후에 나올 수 있는 *e*의 후행항목들이 *b, d, g*임을 알 수 있다. 하지만 (*ceg*)의 경로는 L_2-tree상에 존재하지 않으므로 (그림 6)에서는 점선으로 표현하였다. C_2-matrix를 통해서 *eb*와 *eg*는 빈발하지 않음을 알 수 있으므로 *ceb*와 *ceg*는 빈발여부를 확인하지 않는다. 따라서 (*ced*)를 방문하여 자신의 지지도가 1임을 확인하고 *(ced)*의 분산지지도는 헤더 테이블 HT(c)에 있는 *d*의 동일항목 연결 리스트를 따라가 분산경로 (*cebd*)에서 (*ced*)의 분산지지도 1을 파악하게 되며 (*ced*)의 지지도가 2이므로 빈발 항목집합임을 알 수 있다. (*ced*)의 자식노드에 대한 빈발 항목집합을 탐사하기 위해 C_2-matrix에서 (*ced*)의 자식노도로 가능한 항목은 *d*의 후행항목으로 항목 *g*가 유일하다. 그러나 C_2-matrix를 참고하면 *bg*는 빈발하므로 *cedg*의 빈발여부를 확인하면 분산경로 *(cebdg)*의 지지도는 1이므로 빈발하지 않는다. 재귀적으로 (*ce*)로 돌아오고 (*ce*)는 다시 (*c*)로 돌아가게 된다. 다음의 후보 항목집합으로 C_2-matrix의 순서로는 (*ca*)가 되지만 길이가 2인 (*ca*)는 L_2가 되지 않아 L_2-tree로 표현되지 않았으며, 따라서 (*ca*)로 시작되는 빈발항목이 존재하지 않음을 알 수 있다. 다음으로는 (*cb*)로 이동하여 이와 동일한 방법으로 순회하여 (*cbg*)가 빈발항목임을 확인하며 이후에는 (*cd*)와 (*cg*)로 시작하는 빈발항목을 찾게 된다. (그림 7)은 L_2-traverse 알고리즘을 보여준다.

L_2-tree를 순회하며 빈발 항목집합을 탐사하는 방법은 L_2-tree에 대해서 지지도 기반으로 단지 한번의 깊이 우선 탐사만을 수행하므로 트리 탐사 비용보다는 L_2-tree를 구축하는데 필요한 비용이 주된 비용이라 할 수 있다. 이러한 L_2-traverse의 빈발 항목집합 탐사 형태는 FP-growth와 전혀 다른 형태를 보인다. FP-growth에서 사용하는 데이터 구조인 FP-tree는 L_1을 찾고 L_1으로 구성된 각 트랜잭션에 대해 L_1 항목들을 지지도 내림차순으로 경로를 구성하기 때문에 트리 구축 비용은 상대적으로 적다. 그러나 각 후보 항목집합의 빈발 여부를 파악하기 위해 사용하는 FP-growth 알고리즘은 FP-tree에서 조건 패턴을 기반으로 부분 트리를 동적으로 생성하므로 빈발 항목집합 탐사에서 많은 시간을 소모하게 된다. 이와 달리 L_2-traverse는 한번의 깊이 우선 순회만을 필요로 하므로 희소 데이터 집합뿐만 아니라 L_2-tree의 구축비용이 상대적으로 더 커지는 밀집 데이터 집합에서도 FP-growth보다 효율적으로 빈발 항목집합을 구할 수 있다.

```
입력 : DB와 최소지지도 ξ가 주어졌을 때, L2-tree 생성 알
       고리즘에 기반한 L2-tree의 root node, L1과 C2
       -matrix U
출력 : L3이상의 모든 빈발 항목집합
Procedure L2-traverse ( L1, C2-matrix U, ξ, root)
begin
    curr_node := root;
    if level(curr_node) = 1 then
        if L1[curr_node→item] < ξ then return
        endif
    else if level(curr_node) = 2 then
        if U [firstitem(curr_node), curr_node→item] < ξ then
        return
        endif
    else
        if localsup(node) + distsup(node) < ξ then return
        endif
    endif
    for each child_node of curr_node
       if U [curr_node→item, child_node→item] ≥ ξ then
           L2-traverse( L1, U, ξ, child_node)
       endif
    end
end

Procedure distsup(node)
begin
    distSup := 0
    curr_header_table := header_table(firstitem(node))
    curr_node := startNode(curr_header_table, node→item)
    while curr_node is not null
    begin
         if curr_node is superset of node then
              distSup := distSup + localSup(curr_node)
         endif
         curr_node := next_link(curr_node)
    end
    return distSup
end
```

(그림 7) L_2-traverse 알고리즘

4.2 C_3-traverse 알고리즘

L_2-tree를 구성하는 이유는 L_3 이상의 모든 빈발 항목집합을 구하기 위함이다. 이때 L_3가 되지 않을 L_2를 C_2-matrix에서 미리 제거한다면 L_2-tree 생성시 각 변형 트랜잭션의 후행항목 리스트의 크기를 줄일 수 있다. 따라서 L_2-tree의 크기가 작아지고 적은 메모리 공간으로 보다 빠른 시간에 모든 빈발 항목집합을 탐사 할 수 있다.

[정의 7] (최적화된 C_2-matrix) 최적화된 C_2-matrix U_{opt}는 기존의 C_2-matrix U의 원소 $U[a,b]$가 최소지지도보다 작거나, 혹은 선행항목 *a*와 후행항목 *b*에 대해서 *a*의 후행항목 집합과 *b*의 후행항목 집합간에 중복되는 항목이 없고 *a*의 후행항목 집합과 *a*의 후행항목 집합 중에서 *b*를 제외한 항목들의 후행항목 집합간의 중복되는 항목으로 *b*가 없다면, 해당 원소를 고려 대상에서 제외한다. 이와 같이 C_2-matrix에서 향후 고려할 필요가 없는 항목들을 미리 제거한 것을 최적화된 C_2-matrix U_{opt}라 하며 다음과 같이 정의한다.

$$U_{opt}[a,b] = \begin{cases} -1 & \text{if } (U[a,b] < \xi) \text{ or } ((\gamma_a \cap \gamma_b = \emptyset) \text{ and } (\gamma_a \cap \gamma_{\gamma_a - b} \neq b)), \\ U[a,b] & otherwise. \end{cases}$$ □

선 \ 후	b	c	d	e
a	5	6	2	9
b		2	8	3
c			3	7
d				4

(a) U

선 \ 후	b	c	d	e
a	-1	6	-1	9
b		-1	-1	-1
c			-1	-1
d				-1

(b) 최적화된 U_{opt}

(그림 8) C_2-matrix의 최적화

$\xi=5$, $L_1=\{a,b,c,d,e\}$일 때 (그림 8) (a)와 같이 구해지는 C_2-matrix U에 대한 최적화 과정은 다음과 같다. (그림 8) (a)의 U에서 ab의 지지도는 5이므로 빈발 항목집합이며 L_2-tree에 반영되어야 한다. 따라서 ab를 시작으로 하는 길이가 3인 후보 항목집합 C_3는 b의 각 후행항목에 대해 후보 항목집합 abc, abd, abe가 된다. abc가 빈발 항목집합이 되기 위해서는 ab, bc, ac 모두가 빈발 항목집합이 되어야 한다. 하지만 U에서는 $sup(bc)=2$ 임을 알 수 있으며 bc가 빈발 항목집합이 아니므로 abc는 빈발 항목집합이 될 수 없음을 알 수 있다. 이와 동일한 이유로 항목집합 abd의 경우 ad가 빈발하지 않으며, abe의 경우 be가 빈발하지 않으므로 빈발 항목집합이 될 수 없다. 결국 ab로 시작하는 L_3는 존재하지 않게 된다. 항목집합 ac에 대해서는 C_3 집합이 acd, ace가 되며 ac는 빈발 항목집합이 되지만 cd와 ad가 빈발하지 않기 때문에 acd는 빈발하지 않음을 알 수 있고, ace는 ac와 ae, ce가 모두 빈발함을 알 수 있으며 따라서 ace는 빈발함을 알 수 있다. 이때 C_2-matrix를 참조하면 ae는 L_3가 되지 않으므로 $U[a,e]$가 -1로 될 수 있겠지만, $\gamma_a \cap \gamma_c \ni e$ 이므로 [정의 7]에 의해서 -1이 되지 않는다. 만약 ae를 -1로 한다면, ace라는 빈발 항목집합을 찾을 수 없게 된다. 본 논문에서는 L_2를 먼저 탐사하고 L_3이상의 모든 빈발 항목집합을 구하는 L_2-tree를 생성하는 것이므로 L_3가 될 수 없는 L_2항목집합들을 위한 경로를 L_2-tree에 만들 필요가 없슴을 알 수 있다. (그림 8) (b)의 최적화된 C_2-matrix U_{opt}를 이용할 경우 따라서 [정의 7]에 의해 (그림 8) (a)에서 L_3가 될 수 없는 각 항목집합 xy에 대응되는 $U[x,y]$ 값이 -1이 되어 L_2로 처리되지 않으며 결국 L_2-tree에 만들지 않는다. U_{opt}를 사용하여 변형 트랜잭션 $\overline{T_k}=\{a,b,c,d\}$가 처리될 경우에 ab가 L_3의 부분패턴으로 참가하지 않으므로 변형 트랜잭션 $\overline{T_k}=\{a,c,d\}$과 같아지고, ad도 동일한 이유로 L_3의 부분패턴으로 참가하지 않음을 알 수 있으므로 결국 변형 트랜잭션 $\overline{T_k}=\{a,d\}$와 같아진다. (그림 9)의 C_3-traverse 알고리즘은 L_2-tree 구성하기 전에 C_2-matrix를 최적화하고, 이를 바탕으로 L_2-traverse 알고리즘과 동일한 방법으로 빈발 항목집합을 탐사한다.

```
입력 : DB와 최소지지도 ξ가 주어졌을 때, L2-tree 생성
       알고리즘에 기반한 L2-tree
출력 : L3이상의 모든 빈발 항목집합
Procedure C3-traverse
begin
    scan DB and find L1 using L1
    scan DB and find L2 using C2-matrix U
    opt_C2_matrix( U, ξ)
    root node = gen_L2_tree( U, ξ)
    L2-traverse ( L1, C2-matrix U, ξ, root node)
end

Procedure opt__C2__matrix ( C2-matrix U, ξ)
begin   // 최적화
    for i := 0 to maxL1 - 1
        for j := i + 1 to maxL1
            if U[i, j].count ≥ ξ and U[i, k].pass is not true
then
                flag := true
                for k := j + 1 to maxL1
                    if U[i, k].count ≥ ξ and U[i, k].count
                    ≥ ξ then
                        flag := false
                        U[i, k].pass := true
                    endif
                end
                if flag = true then U[i, j].count := -1
                endif
            endif
        end
    end
end
```

(그림 9) C_3-traverse 알고리즘

5. 실험 및 결과 분석

본 절에서는 L_2-traverse와 C_3-traverse의 효율성 및 확장성을 FP-growth[10] 알고리즘 및 H-mine 알고리즘[14]과 비교하여 검증한다. 모든 실험은 500MB의 메인 메모리를 가진 1.7-GHz 펜티엄 PC와 Linux 운영체제에서 수행되었다. 본 논문에서 제시된 실행 시간은 프로그램의 총 수행시간으로 CPU 사용시간 뿐만 아니라 알고리즘 수행에 필요한 디스크 접근 시간을 포함하여 Input/Output와 관련된 모든 시간이 포함된다. 메모리 사용량은 Linux 운영체제가 proc파일 시스템을 통해 제공하는 프로세스 메모리 사용량으로 비교하였다. 본 논문에서는 <표 2>에서와 같이 서로 다른 밀집도를 갖는 두 개의 데이터 집합을 실험하였으며, 대부분의 빈발 항목집합 탐사 방법들의 성능 비교에 활용되는 표준 데이터 생성 프로그램[9]을 사용하여 생성하였다. <표 3>

〈표 2〉 실험에 사용한 데이터 집합

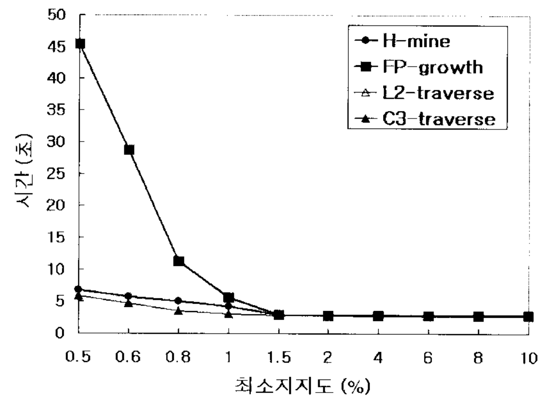
	트랜잭션 개수	트랜잭션당 평균 항목 수	잠재적인 최대 빈발 항목집합의 평균 크기	*DB* 생성에 사용된 항목의 수
D_1	100,000	25	20	10,000
D_2	100,000	15	10	1,000

〈표 3〉 길이별 빈발항목 수 (최소지지도 = 0.5%)

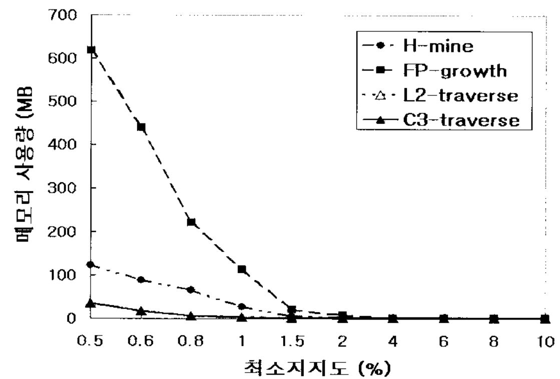
	$\|L_1\|$	$\|L_2\|$	$\|L_3\|$	$\|L_4\|$	$\|L_5\|$
D_1	6,640	0	x	x	x
D_2	728	317	3	x	x

3>은 이들 데이터의 밀집도를 비교하기 위한 것으로서, 본 논문의 실험에서 사용된 가장 작은 최소지지도인 0.5%로 마이닝을 수행하였을 때 각 길이별 빈발항목의 수를 나타낸다.

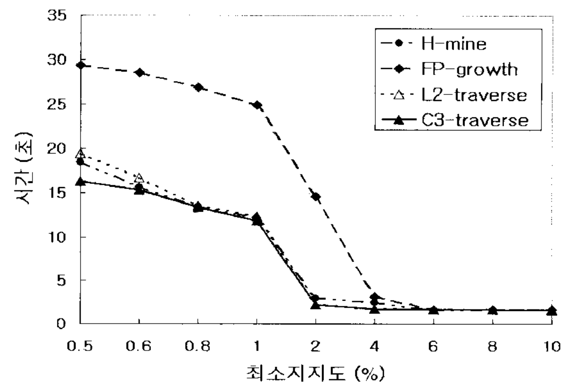
(그림 10)에서는 두 데이터 집합 중에서 보다 희소 데이터 집합인 D_1에 대하여 빈발 항목집합 탐사를 위한 수행시간을 비교하였다. 일반적으로 희소 데이터 집합에서의 빈발 항목집합 탐사 과정에서는 충분히 긴 길이를 갖는 빈발 항목집합의 개수가 적으며, 따라서 의미 있는 규칙을 찾기 위해 사용자는 일반적으로 매우 낮은 최소지지도를 설정할 수 있다. 따라서 본 실험에서도 매우 상당히 낮은 지지도 범위에서 실험되었다. 그림에서 FP-growth는 1%에서부터 실행시간이 급격히 증가하는 것을 볼 수 있다. 이는 데이터 집합 D_1은 1%와 1.5% 사이에서 다수의 L_1들이 생성되며, 이보다 큰 최소지지도에서는 L_1이 거의 변하지 않기 때문이다. 제안된 방법들의 수행시간은 FP-growth에 비해서는 월등히 적으며 H-mine에 비해서는 다소 작음을 알 수 있다. 이러한 결과로부터 본 논문에서 제안된 방법들이 희소 데이터 집합에 대하여 낮은 지지도에서 의미 있는 빈발 항목집합을 찾는 작업에 있어서 매우 효율적임을 알 수 있다. 한편, 본 실험에서 L_2-traverse와 C_3-traverse에는 성능차이가 없는 이유는 두 알고리즘의 차이점이 C_2-matrix의 최적화를 통한 L_2-tree의 생성방법에 있으나 D_1에서는 L_2가 존재하지 않아 L_2-tree를 구성하지 않기 때문이다. 따라서 두 방법이 서로 동일한 단계에서 수행을 종료하게 되므로, 수행 시간에 차이가 발생되지 않는다. (그림 11)은 D_1에 빈발 항목집합 탐색 과정에서 메모리 사용량을 비교한 것이다. 해당 실험 결과에서도 본 논문에서 제안된 두 방법들이 FP-growth에 비해 메모리 사용량이 월등히 적으며, H-mine에 비해서도 상대적으로 매우 작은 메모리 공간을 사용함을 알 수 있다. 이러한 결과로부터 본 논문에서 제안된 방법들이 희소 데이터 집합에 대하여 낮은 지지도에서 의미 있는 빈발 항목집합을 찾는 작업에 있어서 수행 메모리 사용량 측면에서도 우수한 성능을 보임을 알 수 있다.



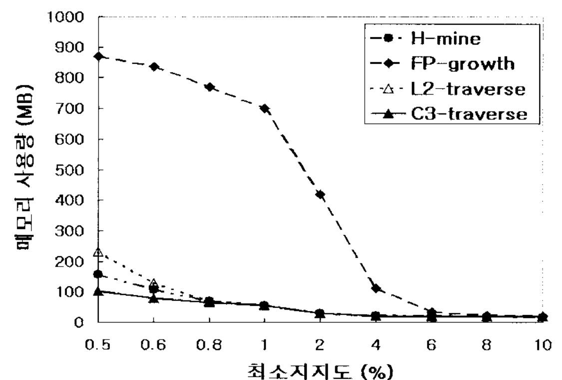
(그림 10) D_1의 실행 시간



(그림 11) D_1의 메모리 사용량



(그림 12) D_2의 실행시간

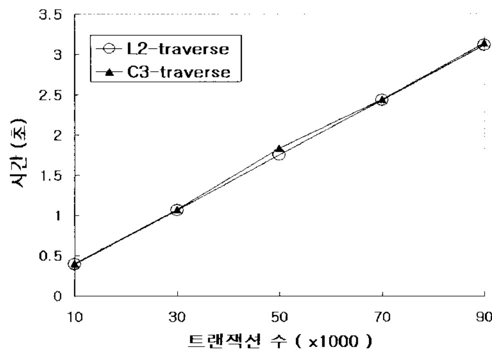


(그림 13) D_2의 메모리 사용량

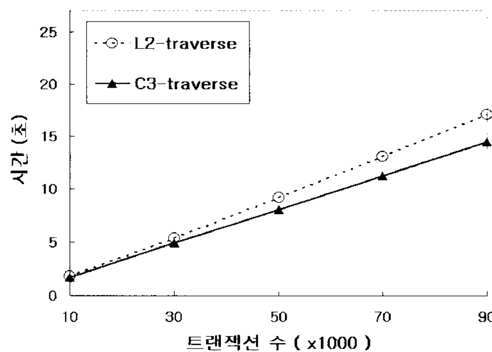
D_2는 희소 데이터 집합이기는 하나 D_1과에 비해서는 상대적으로 밀집 데이터 집합이다. (그림 12)는 D_2에 대한 수행 시간을 보여준다. 본 논문에서 제안된 두 가지 방법 중에서 L_2-traverse 방법의 수행 시간은 H-mine의 수행 시간에 비해서 다소 크나 C_3-traverse 방법의 수행 시간은 H-mine 방법과 거의 유사하거나 다소 작음을 알 수 있다. 한편, C_2-matrix의 최적화 효과로 최소지지도가 작아질수록 L_2-traverse와 C_3-traverse의 실행시간 차이가 커짐을 볼 수 있다. 메모리 사용량을 나타내는 (그림 13)에서도 최적화의 효과로 최소지지도 0.6%와 0.8% 사이에서 L_2-traverse와 C_3-traverse의 메모리 사용량이 차이가 발생됨을 알 수 있다. 메모리 사용량 측면에 있어서도 수행 시간에서와 마찬가지로 L_2-traverse 방법은 H-mine에 비해서 다소 크나 C_3-traverse 방법은 H-mine과 거의 유사하거나 다소 작음을 알 수 있다.

이러한 실험 결과들로부터 본 논문에서 제안된 방법들이 희소 데이터 집합에서는 FP-growth 방법 및 H-mine 방법 등과 같은 기존의 다른 방법들에 비해 수행 시간 및 메모리 사용량 측면에서 우수한 성능을 보임을 알 수 있다. 이러한 경향은 마이닝 수행 대상 데이터 집합의 희소율이 높아질수록 커진다. 즉, 보다 희소 데이터 집합일수록 본 논문에서 제안된 방법들은 보다 효율적으로 빈발 항목집합을 탐사한다.

제안된 알고리즘에 대해 트랜잭션 개수에 따른 확장성을 평가하기 위해서 (그림 14) (a)는 최소지지도를 0.8%로 설정하고 D_1의 매개변수 중 트랜잭션 개수를 10K에서 90K로 확장하면서 수행된 실험 결과를 보여주며, (그림 14) (b)는 최소지지도를 0.5%로 설정하고 D_2의 매개변수 중 트랜잭션 개수를 10K에서 90K로 확장하면서 수행된 실험 결과를 보여준다. 트랜잭션의 늘어감에 따라 모든 방법들에서 실행시간이 선형적으로 증가함을 볼 수 있다. 한편, (그림 14) (a)에서 L_2-traverse와 C_3-traverse의 실행 시간이 차이가 없는 이유는 앞서 기술한 바와 같이 D_1에서 L_2가 생성되지 않기 때문이다. 반면 (그림 14) (b)의 결과에서는 D_2에서 L_2가 생성되므로 최적화 수행 여부에 따라 L_2-traverse와 C_3-traverse가 성능 차이를 보인다.



(a) D_1



(b) D_2

(그림 14) 트랜잭션 개수에 따른 확장성

일반적으로 데이터 집합의 희소성은 상대적으로 평가될 수 있다. 본 실험 결과에서 사용된 두 데이터 집합은 상대적인 희소도의 차이를 고려하여 실험 대상 데이터 집합으로 선택하였으며, 만약 D_1보다 희소한 데이터 집합을 대상으로 실험하는 경우 빈발항목 탐색 과정에서의 보다 나은 성능을 보일 것으로 판단된다.

6. 결 론

기존의 연관규칙 탐사 알고리즘은 크게 Apriori 기반 알고리즘과 성장-패턴 기법으로 구분 할 수 있다. Apriori 기반 알고리즘은 빈발 항목집합이 길수록 수행 시간 및 메모리 사용량이 급격히 증가하는 단점을 보이며, 성장-패턴 기법을 사용한 FP-growth는 희소데이터 집합에서 역시 시간과 공간이 크게 증가한다. 본 논문에서는 빈발 항목집합 탐사를 위해 길이 2인 빈발 항목집합 L_2 기반 데이터 구조를 제안하고, 이를 이용하여 지지도 내림차순으로 깊이 우선 탐사를 단 한번 수행하는 빈발 항목집합 탐사방법을 제안하였다. 일반적으로 희소 데이터 집합에서는 긴 빈발 항목집합이 빈번하게 생성이 되지 않으므로 L_2를 기반으로 한 L_2-tree는 희소데이터 집합에서 L_1에 기반 하는 데이터 구조보다 적은 메모리 공간을 사용한다. L_2-tree를 사용하여 빈발 항목집합을 탐색하는 알고리즘인 L_2-traverse 알고리즘은 FP-growth와 달리 별도의 추출 구조를 만들지 않고 L_2-tree에 대한 오직 한번의 지지도 기반 깊이 우선 순회함으로써 길이 3 이상의 모든 빈발 항목집합을 찾는다. 또한, L_2-traverse에 대한 최적화 방법으로 C_3-traverse 알고리즘을 제안하였다. 이 알고리즘은 L_2-traverse와 거의 유사하나 L_3를 구성할 수 없는 L_2를 C_2-matrix에서 미리 제거함으로써 L_2-tree의 메모리 사용량과 빈발 항목집합 탐사를 위해 필요한 순회 시간을 줄인다. 이러한 최적화의 효과로 메모리 공간을 10~50%, 수행시간은 10~70%정도의 향상됨을 실험을 통하여 검증하였다. 특히, 희소데이터 집합에서 L_2-traverse 와 C_3-traverse는 FP-growth보다 매우 적은 양의 메모리를 필요로 하며 수행 시간도 매우 작

으며, H-mine 방법에 비해서도 메모리 사용량 및 수행 시간 측면에서 거의 동일하거나 최소 지지도 값에 따라 다소 우수한 성능을 나타낸다.

참 고 문 헌

[1] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. "Finding interesting rules from large sets of discovered association rules," In *Proc. of the 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, MD, pp.401-408, November, 1994.

[2] B. Lent, A. Swami, and J. Widom. "Clustering association rules," In *Proc. of the Int. Conf. on Data Engineering*, Birmingham, England, pp.220-231. April, 1997.

[3] J.S. Park, M.S. Chen, and P.S. Yu. "An effective hash-based algorithm for mining association rules," In *Proc. of the ACM-SIGMOD Int. Conf. on Management of Data*, San Jose, CA, pp.175-186, May, 1995.

[4] S. Sarawagi, S. Thomas, and R. Agrawal. "Integrating association rule mining with relational database systems: Alternatives and implications," In *Proc. of the ACM-SIGMOD Int. Conf. on Management of Data*, Seattle, WA, pp.343-354, June, 1998.

[5] R. Srikant and R. Agrawal. "Mining generalized association rules," In *Proc. of the Int. Conf. on Very Large Data Bases*, Zurich, Switzerland, pp.407-419, September, 1995.

[6] R. Srikant, Q. Vu, and R. Agrawal. "Mining association rules with item constraints," In *Proc. of the 3rd Int. Conf. on Knowledge Discovery and Data Mining*, Newport Beach, CA, pp.67-73, August, 1997.

[7] R. Agarwal, C. Aggarwal, and V.V.V. Prasad. "Depth first generation of long patterns," In *Proc. of the 6th Int. Con. Knowledge Discovery and Data Mining*, pp.108-118, August, 2000.

[8] C. Hidber, "Online Association Rule Mining", In *Proc. of the ACM-SIGMOD Int. Conf. on Management of Data*, Philadelphia, PA, pp.145-156, May, 1999.

[9] R. Agrawal and R. Srikant. "Fast algorithms for mining association rules," In *Proc. of the Int. Conf. on Very Large DataBases*, Santiago, Chile, pp.487-499, September, 1994.

[10] J. Han, J. Pei, and Y. Yin. "Mining frequent patterns without candidate generation," In *Proc. of the ACM-SIGMOD Int. Conf. on Management of Data*, Dallas, TA, pp.1-12, May, 2000.

[11] R. Agarwal, C. Aggarwal, and V.V.V. Prasad. "A tree projection algorithm for generation of frequent itemsets," In *Journal of Parallel and Distributed Computing*, Vol.61, No. 3, pp.350-371, 2001.

[12] A. Savasere, E. Omiecinski, and S. Navathe. "An efficient algorithm for mining association rules in large databases", In *Proc. of the Int. Conf. on Very Large DataBases*, Zurich, Switzerland, pp.432-443, September, 1995.

[13] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. "Dynamic itemset counting and implication rules for market basket analysis," In *Proc. of the ACM-SIGMOD Int. Conf. on Management of Data*, Tucson, AZ, pp.255-264, May, 1997.

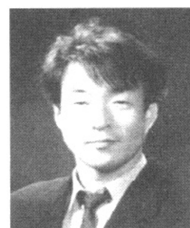
[14] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases", In *Proc. of the Int. Conf. on Data Mining*, San Jose, CA, pp.441-448, November, 2001.

박 인 창

e-mail : inchang.park@samsung.com
2001년 안양대학교 컴퓨터공학과(학사)
2003년 연세대학교 대학원 컴퓨터과학과 (석사)
2003년~현재 삼성전자 선임연구원
관심분야 : 데이터 마이닝, OLAP, 데이터웨어하우징, 정보추출

장 중 혁

e-mail : jhchang@amadeus.yonsei.ac.kr
1996년 연세대학교 컴퓨터과학과(학사)
1998년 연세대학교 대학원 컴퓨터과학과 (석사)
2005년 연세대학교 대학원 컴퓨터과학과 (박사)
2005년~현재 연세대학교 소프트웨어응용연구소 전문연구원
관심분야 : 데이터 스트림, 데이터 마이닝, 정보보안, 생물정보학



이 원 석

e-mail : leewo@amades.yonsei.ac.kr
1985년 미국 보스턴대학교 컴퓨터과학과 (학사)
1987년 미국 퍼듀대학교 컴퓨터공학과 (석사)
1990년 미국 퍼듀대학교 컴퓨터공학과 (박사)
1990년~1992년 삼성전자 선임연구원
1993년~1999년 연세대학교 컴퓨터과학과 조교수
1999년~2004년 연세대학교 컴퓨터과학과 부교수
2004년~현재 연세대학교 컴퓨터과학과 교수
관심분야 : 분산 데이터베이스, 멀티미디어 데이터베이스, 객체지향 시스템, 데이터마이닝