완전해싱을 위한 DHP 연관 규칙 탐사 알고리즘의 개선 방안

(Improvement of DHP Association Rules Algorithm for Perfect Hashing)

이 형 봉 †

(Hyung Bong Lee)

요 약 DHP 연관 규칙 탐사 알고리즘은 후보 빈발 항목 집합들에 대한 계수를 유지하기 위한 해쉬 트리의 크기를 가능한 한 줄이기 위하여, 독립된 직접 해쉬 테이블을 미리 준비해 놓았다가 후보 빈발 항목 집합을 생성할 때 적용함으로써 전지 효과를 얻는다. 이 때 직접 해쉬 테이블의 크기가 클수록 전지 효과는 커지며, 특히 길이 2인 후보 빈발 항목 집합을 생성하는 단계에서의 전지 효과는 알고리즘 전체의 성능을 좌우할 만큼 큰 영향을 발휘한다. 따라서 급속도로 보편화되고 있는 대용량 주기억 장치 시스템 추세에 따라 단계 2에서의 직접 해쉬 테이블 크기의 극단적인 증가에 대한 시도가 이루어지고 있으며, 이러한 것 중의 하나가 완전 해쉬 테이블이다. 그러나 단계 2에서의 완전 해쉬 테이블을 사용할 경우, 이를 단순히 기존 DHP 알고리즘에 적용하여 버켓 크기($|H_2|$)만을 재설정하는 것 보다, DHP 알고리즘 자체를 조금 변경했을 때 약 20% 이상의 추가 성능 이득을 얻을 수 있음이 밝혀졌다. 이 논문에서는 단계 2에서의 완전 해쉬 테이블의 타당성을 조명해 본 후, 그 특성을 충분히 활용하도록 DHP를 개선한 PHP 알고리즘을 제안하며 그 결과를 실험적 환경에서 검증한다.

키워드 : DHP, 대용량 주기억 장치, 해쉬 트리, 직접 해쉬 테이블, 완전 해쉬 테이블, PHP

Abstract *DHP* mining association rules algorithm maintains previously independent direct hash table to reduce the size of hash tree containing the frequency number of each candidate large itemset. It performs pruning by using the direct hash table when the hash tree is constructed. The more large the size of direct hash table increases, the higher the effect of pruning becomes. Especially, the effect of pruning in phase 2 which generate 2-large itemsets is so high that it dominates the overall performance of *DHP* algorithm. So, following the speedy trends of producing VLM(Very Large Memory) systems, extreme increment of direct hash table size is being tried and one of those trials is perfect hash table in phase 2. In case of using perfect hash table in phase 2, we found that some rearrangement of DHP algorithm got about 20% performance improvement compared to simply $|H_2|$ reconfigured DHP algorithm. In this paper, we examine the feasibility of perfect hash table in phase 2 and propose *PHP* algorithm, a rearranged DHP algorithm, which uses the characteristics of perfect hash table sufficiently, then make an analysis on the results in experimental environment.

Key words : DHP, VLM, hash tree, direct hash table, perfect hash table, PHP

1. 서 론

연관 규칙 탐사 알고리즘들은 전통적으로 후보 빈발 항목 집합을 계수하기 위하여 해쉬 트리를 사용한다[1]. 이 때 해쉬 트리는 거래에 포함된 부분 항목 집합이 존재하는지의 여부를 판별하는 검색과 존재가 확인될 경우 계수를 유지하는 등의 두 가지 역할을 담당한다. 이는 해쉬 트리를 구성하는 후보 빈발 항목 집합 수의 크기에 따라 연관 규칙 탐사 알고리즘의 성능이 영향을 받을 수 있으며, 따라서 가능한 후보 빈발 항목집합의 수를 줄여야 함을 의미한다.

DHP 연관 규칙 탐사 알고리즘[2]은 후보 빈발 항목 집합 수를 최소화 시키기 위하여 독립된 직접 해쉬 테이블(direct hash table)을 미리 준비해 놓았다가 후보 빈발 항목 집합을 생성할 때 적용함으로써 전지 효과를 얻는다. 이 때 직접 해쉬 테이블은 간단한 해쉬 함수에 의해 관리되므로 큰 부담이 되지 않으며, 해쉬 테이블의

† 종신회원 : 강릉대학교 컴퓨터공학과 교수
hblee@kangnung.ac.kr
논문접수 : 2003년 9월 29일
심사완료 : 2003년 11월 26일

크기가 클수록 전지 효과는 커진다. 특히 길이 2인 후보 빈발 항목 집합은 그 개수가 많을 뿐 아니라, 거래에 포함된 검색 대상이 되는 부분 항목 집합들이 많으므로, 이 단계에서의 전지 효과는 알고리즘 전체의 성능을 좌우할 만큼 지대하다[2].

그러므로 *DHP* 알고리즘의 성능을 향상시키기 위해 직접 해쉬 테이블의 크기를 늘리려는 시도는 당연한 일인데, 이 때 시스템 메모리의 크기를 고려해야 하는 문제가 있다.

이 논문에서는 최근 보편화되고 있는 대용량 주기억장치(VLM, Very Large Memory) 시스템의 보편화에 따라 *DHP* 알고리즘의 단계 2에서 직접 해쉬 테이블의 크기를 극단적으로 증가시킨 완전 해쉬 테이블(perfect hash table)의 실현 가능성을 타진해보고, 완전 해쉬 테이블의 특성을 충분히 활용하도록 *DHP*를 개선한 *PHP* (Perfect Hashing and Pruning) 알고리즘을 제안한다. 이를 위하여 2 장에서 관련 연구, 3장에서 *DHP* 알고리즘의 성능 특성과 VLM 시스템, 4장에서 완전 해싱을 위한 *DHP* 알고리즘의 개선에 대하여 각각 논하고 마지막으로 5장의 결론으로 이 논문을 맺는다.

2. 관련연구

2.1 연관 규칙 탐사 알고리즘 관련 용어

연관 규칙 탐사 알고리즘은 대량의 거래 자료로부터 거래 항목들 사이에 존재하는 동시 거래 관련성을 발견하는 것으로, 지지도(support), 신뢰도(confidence), 항목 집합(itemset), 빈발 항목 집합(large itemset), 선두 빈발 후보 항목 집합(frontier large itemset), 후보 항목 집합(candidate large itemset), 최대 빈발 항목 집합 (maximal large itemset), 길이가 k인 항목 집합 (k-item set), 전체 항목 집합 등의 용어와 표 1의 기호들을 사용한다[1-4,6].

표 1 연관 규칙 탐사에서 사용되는 기호

D_k, t_k, t	단계 k에서의 데이터베이스, $t_k, t \in D$
L_k, l_k, l	단계 k에서의 {빈발 항목 집합}, $l_k, l \in L_k$
F_k, f_k, f	단계 k에서의 {선두 빈발 항목 집합}, $f_k, f \in F_k$
C_k, c_k, c	단계 k에서의 {후보 빈발 항목 집합}, $c_k, c \in C_k$
H_k, h_k	단계 k에서의 직접 해쉬 테이블, 해쉬 함수
I, i_j	전체 항목 집합, $i_j \in I$
Nnnnn	전체 항목 집합의 크기($\|I\|$)가 *nnnn* 임
TxIyDz	거래의 평균 크기 x, 빈발 항목 집합의 평균 크기 y, 총 거래 건수 $z*1000$개인 거래 데이터

2.2 연관 규칙 탐사 알고리즘의 진화과정

연관 규칙 탐사를 위한 가장 원시적인 알고리즘은 전체 거래 항목 집합 I의 모든 부분 집합에 대한 계수공간을 마련 한 후, 각 거래로부터 가능한 모든 부분 집합을 도출하여 계수함으로써 모든 빈발 항목 집합을 한번의 DB 스캔으로 찾을 수 있다. 그러나 부분 집합 및 계수를 위한 저장 공간이 엄청나게 요구되므로 비현실적이다.

*AIS*는 단계별로 L_k, F_k, C_k를 생성하고 계수하는 다단계 알고리즘이다[4]. 이 알고리즘은 바로 전 단계에서 생성된 $F_{k-1}(F_0=\{\emptyset\})$를 각 거래를 읽는 도중에 L_k의 가능성이 있는 것을 C_k로 즉흥적으로 확장하고 계수한다. F_{k-1}의 확장은 거래에 포함된 모든 참조 대상 항목을 하나씩 추가하면서 이루어지고, 그 과정은 확장 대상인 f_{k-1}의 지지도, 추가 예정 항목의 현재까지의 상대 지지도 등을 고려하여 빈발하지 않을 것으로 예측되는 첫 항목까지만 계속된다. F_k는 C_k에서 빈발 항목 집합이 아닐 것으로 예측되었던 것이 실제로는 l_k로 판명된 c_k들만으로 구성된다. 따라서 f_{k-1}, c_k, l_k의 길이는 한결같지 않고 다양하다. *AIS* 알고리즘은 DB를 여러 번 스캔함으로써 각 단계별로 계수를 위한 메모리 양을 줄였다는 데 의의가 있다.

Apriori 알고리즘[1]은 각 단계에서 C_k를 미리 고정시키고($C_0=I$) DB를 스캔하여 계수한 후 L_k를 결정한다. 이 때 C_k는 L_{k-1}로부터 정확히 하나의 항목만이 확장되어 생성되므로 c_k, l_k는 모두 길이 k인 항목 집합이다. 이 알고리즘은 C_k의 관리를 위하여 해쉬 트리를 사용하였는데, *AIS*에 비하여 DB 스캔 회수는 늘어나지만 각 단계에서의 C_k에 대한 검색 및 계수과정이 효과적이어서 전체적인 성능이 월등히 높다.

SETM[5]은 관계 연산 즉, SQL 문장으로 해결하는 방안을 시도하였고, *Partition* 알고리즘[6]은 *Apriori*를 바탕으로, 전체 DB를 메모리에 적재할 수 있을 만큼의 크기로 분할하여 각각의 분할에 대하여 한 번의 DB 스캔으로 빈발 항목 집합을 도출한 후, 그들을 병합하여 전체 빈발 항목 집합으로 결정하기 위하여 전체 DB를 한 번 더 스캔한다. 즉, 이 알고리즘은 두 번의 DB 스캔으로 연관 규칙 탐사를 마친다.

DHP 알고리즘[2]은 *Apriori*에서 C_k 대한 탐색 및 계수를 위한 성능을 개선하기 위한 두 가지 방안을 도입하였다. 첫 번째 방안으로 C_k 자체의 크기를 최대한 줄이기 위하여 바로 전 단계에서, c_k로 예상되는 부분 항목 집합들의 지지도를 H_k에 저장해 놓았다가 C_k를 생성할 때 적용하여 전지한다. 두 번째로는 매 단계에서 t_{k-1}로부터 지지도에 기여하지 못하는 항목들을 제거하여 D_k의 크기를 줄임으로써 불필요한 입·출력 및 C_k 탐색을 제거한다.

CHT 알고리즘[7]은 *DHP*에서 매 단계마다 DB 스캔

을 해야하는 입·출력 부담을 줄이기 위해, $C_k \sim C_{k+i}$를 미리 예측하여 여러 단계를 한 번의 DB 스캔으로 계수한다. 이 알고리즘은 DB 스캔 회수가 줄어드는 반면, 정교하지 못한 $C_k \sim C_{k+i}$의 크기가 증가하므로 해쉬 트리에 대한 검색 및 계수 비용이 늘어나는 대가를 치른다.

위에서 언급한 여러 알고리즘들의 성능은 DB의 크기나 특성, 지지도, 거래 항목 수 등 다양한 요인에 영향을 받게 되므로 절대적으로 평가될 수는 없다. 즉, 어떤 극단적인 상황에서는 원시 알고리즘이 가장 우수한 성능을 발휘할 수도 있다는 것이다. 그러나 보편적인 상황에서 대부분의 연관 규칙 탐사 알고리즘은 *DHP*를 근간으로 하고있다.

또한 위 알고리즘들은 주로 C_k 및 D_k의 축소와 DB 스캔 회수 절약에 초점을 두고 있으며 해쉬 전략 자체에 대해서는 자세히 다루지 않는다.

3. DHP 알고리즘의 성능 특성과 VLM 시스템

그림 1의 *DHP* 연관규칙 탐사 알고리즘의 가장 큰 특징은 미리 계산된 H_k에 의해 해쉬 트리에 저장될 C_k의 크기를 줄인다는 점에 있다. 이 때, H_k에서의 충돌이 크면 정확도가 떨어지므로 전지율이 낮아진다. 따라서 $|H_k|$가 클수록 전지율이 높아지고 성능은 향상되므로, 가능하다면 충돌이 없는 완전 해싱이 바람직하다.

3.1 DHP 알고리즘의 성능 특성 분석

그림 2에 표 2와 표 3에 따른 다양한 형태의 시험용 데이타[3]에 대하여 버켓 크기(모든 $|H_k|$에 동일하게 적용)를 변화시키면서 표 4의 시스템에서 측정한 DHP알고리즘의 성능 추이를 보였다(그림 2의 (a)와 (c)에서는 편의상 단계 2의 실행 시간 측정 값에 350을 일률적으로 가산하였음). 여기서 표 2의 데이타 유형은 표 3에 보인 네 가지 거래 환경을 대표할 수 있도록 구성하였다.

그림 2를 분석하면 다음 내용들을 추론할 수 있다.

- $|H_k|$가 커질수록 전체적인 성능이 높아지다가, 어느 시점을 지나면 오히려 성능이 악화된다. 그 이유는 해싱의 적중률 및 전지율 향상에 따른 성능 이득보다는 해쉬 테이블의 초기화와 메모리 관리를 위한 운영체제의 부담이 더 커지기 때문이다.

표 2 *DHP* 성능 특성 시험에 사용된 DB 규격

[illegible]	[illegible]	[illegible]
T20I6D100 (N1000)	0.5%	697, 5834, 4205, 4933, 4800, 3543, 2024, 855, 267, 29
	0.75%	620, 2162, 751, 702, 481, 319, 159, 54, 11, 1
T20I6D100 (N7000)	0.5%	1474, 2573, 3959, 4918, 4740, 3469, 1945, 771, 195, 8
	0.75%	933, 613, 655, 568, 470, 319, 159, 54, 1

```
s = a minimum support;                    /* - Part 1 - */
set all buckets of H2 to zero;            /* hash table */
forall transactions t∈D do begin
  insert & count 1-items occurrences in hash tree;
  forall 2-subsets x of t do
     H2[h2(x)]++;
end
L1 = {c|c.count≥ s, c is s leaf node of hash tree};
---------------------------------------------------------
k = 2;  Dk = D;                           /* - Part 2 - */
while(|{x|Hk[x] ≥ s}| ≥ LARGE) {
  gen_candidate(Lk-1, Hk, Ck); /* make hash table */
  set all the buckets of Hk+1 to zero;
  Dk+1 = Ø;
  forall transactions t∈D do begin
    count_support(t, Ck, k, t'); /* t'⊆t */
    if (|t'| > k) then do begin
       make_hasht(t', Hk, k, Hk+1, t'');
       if (|t''| > k) then Dk+1 = Dk+1 ∪{t''};
    end
  end
  Lk = {c∈Ck|count≥ s};
  k++;
}
---------------------------------------------------------
gen_candidate(Lk-1, Hk, Ck);              /* - Part 3 - */
while(|Ck| > 0) {
  Dk+1 = Ø;
  forall transactions t∈D do begin
    count_support(t, Ck, k, t'); /* t'⊆t */
    if (|t'| > k) then Dk+1 = Dk+1 ∪{t'};
  end
  Lk = {c∈Ck|count≥ s};
  if ((|Dk| = 0) then break;
  Ck+1 = apriori_gen(Lk); k++;
}
---------------------------------------------------------
Procedure gen_candidate(Lk-1, Hk, Ck)
  Ck = Ø;
  forall c = cp[1]·... ·cp[k-2]·cp[k-1]·cq[k-1],
            cp, cq ∈ Lk-1, |cp ∩ cq| = k-2  do
    if (Hk[hk(c)] ≥ s) then
       Ck = Ck ∪{c}; /* insert c into hash tree*/
End Procedure
---------------------------------------------------------
Procedure make_hasht(t', Hk, k, Hk+1, t'')
  forall (k+1)-subsets x (=t'i1...t'ik) of t' do
    if (for all k-subsets y of x, Hk[hk(y)] ≥ s)
      then do begin
        Hk+1[hk+1(x)]++;
        for (j = 1; j ≤ k+1; j++) a[ij]++;
      end
  for (i=0, j=0; i < |t'|; i++)
    if (a[i] > 0) then do begin
      t''j = t'i; j++;
    end
End Procedure
```

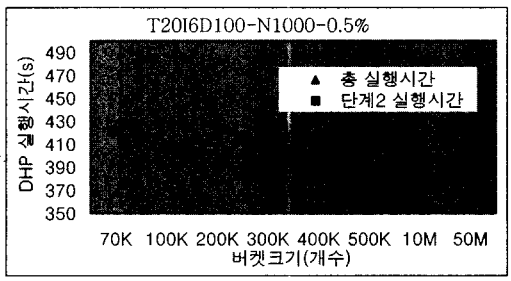
그림 1 DHP 연관 규칙 탐사 알고리즘

표 3 표 2의 시험 데이타 분류 기준

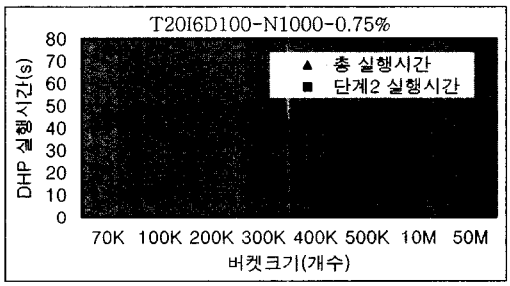
항목 / 빈발집합	조밀	희박
많음	T20I6D100, N1000-0.5%	T20I6D100, N7000-0.5%
적음	T20I6D100, N1000-0.75%	T20I6D100, N7000-0.75%

표 4 DHP 알고리즘 성능시험 시스템 환경

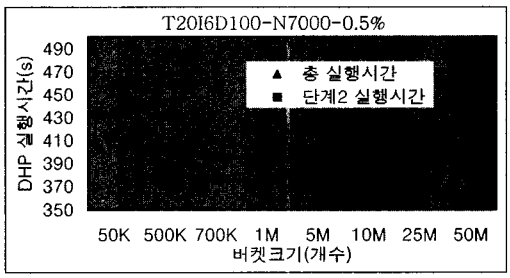
항 목	사 양
모 델	COMPAQ WS au600
C P U	Alpha Ev67 667MHz
메 모 리	1GB
운 영 체 제	Digital Tru64UNIX 4.0F



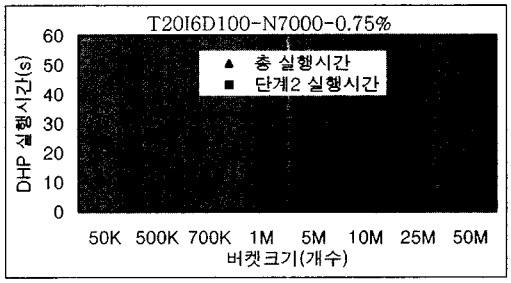
(a)



(b)



(c)



(d)

그림 2 표 2의 데이타에 대한 *DHP*의 성능

- 성능 향상 구간에서는 거래 항목이 조밀하고 빈발항목 집합이 많을수록 성능 이득이 뚜렷하다. 이는 거래 항목의 조밀성에 따라 해싱의 적중 효과가 높아지고 전지되는 빈발 항목 집합이 상대적으로 많아지므로 해쉬 트리에 대한 검색 및 계수 부담의 감소 폭이 커지기 때문이다.
- 단계 2의 성능은 꾸준히 향상되다가 이 단계에서의 완전 해쉬 테이블 크기를 지나면 성능 향상이 멈춘다. 이는 단계 2에서 사용하는 최대 메모리가 완전 해싱 부분까지이고, 나머지 부분은 접근하지 않기 때문이다.

위 사실들은 거래 데이타 및 시스템 환경에 따라서 단계 2에 집중한 *DHP* 알고리즘의 성능 향상이 전체 성능을 높일 수 있는 하나의 방안이 될 수 있음을 의미한다. 단계 2의 성능을 최대로 향상시키는 방법은 그림 1의 line 5~6에서 H_2를 완전 해쉬 테이블로 구현하는 것이다. 이를 위해서는 최소한 ($\frac{N}{2}$)개의 계수 공간이 필요하다. 예를 들어 N이 1,000이라면 약 2MB(sizeof(int)*1000*999/2)가 필요하고, N이 7,000이라면 100 MB 정도가 필요하다. 그림 2에서는 모든 단계에서 버켓 크기를 동일하게 적용하였고, H_k가 얻어질 때까지 H_{k-1}이 유지되어야 하므로 버켓 크기가 50M개인 경우 약 400MB(sizeof(int)* 50M*2)의 메모리가 필요하다.

메모리 소자의 고집적화 및 가격 하락과 프로세서의 발전으로 메모리 비용이 크게 줄어든 현대 컴퓨터에서 2MB는 아주 적은 양이고, 100MB나 400MB 또한 무모할 정도로 큰 양은 결코 아니다.

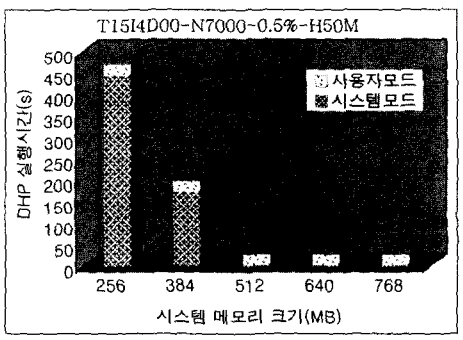
3.2 VLM(Very Large Memory) 시스템과 운영체제

컴퓨터 시스템의 메모리 용량은 일차적으로 프로세서의 특징에 의해 제한을 받는다. 프로세서와 메모리는 데이타 버스, 주소 버스, 제어 버스 등 세가지 버스로 연결되는데, 이 때 주소를 싣는 주소 버스의 크기가 메모리 최대 용량에 결정적인 영향을 미친다. 32비트 프로세서의 경우 4GB(2^{32}) 이상의 메모리는 지원할 수 없다.

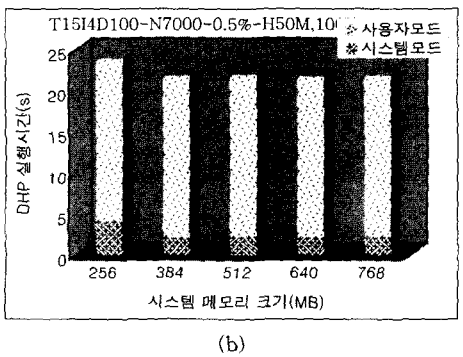
VLM 시스템은 전통적으로 많이 사용되었던 32비트 프로세서의 한계를 극복한 시스템을 말하는데, 최초로 상용화된 64 비트 프로세서인 Alpha와 이를 지원하는 운영체제 Digital UNIX(Tru64 UNIX)가 등장하면서 대두된 개념이다[8]. 32비트 프로세서의 주소 공간이 4GB이긴 하지만 사용자 영역과 시스템 영역의 분리, 입·출력을 위한 하드웨어 주소 영역, 그 밖에 설계상의 이유 등으로 실제로는 1GB를 최대로 하는 경우가 대부분이다. 따라서 상용화된 VLM 시스템은 작게는 1GB, 많게는 16GB이상의 대용량 메모리가 장착되고, 이러한 VLM의 이점을 충분히 활용하려는 독특한 DBMS기술들이 제안되기도 하였다[9].

또한, 운영체제는 응용 프로그램에게 물리 메모리를 바탕으로 주소 공간 및 스왑 영역이 허용하는 범위 내에서 가상 메모리를 제공하기 때문에, *DHP*와 같은 응용 프로그램은 거의 무한대의 메모리를 할당하여 사용할 수 있다. 다만 물리 메모리에 비하여 응용 프로그램에 지나치게 많은 메모리가 할당될 경우 스와핑에 따른 운영체제의 부담이 늘어난다[10].

물리 메모리의 크기가 운영체제의 부담에 미치는 영향을 살펴보기 위해, T15I4D100, N7000, 지지도 0.5%인 거래 데이타에 대해서 모든 단계의 버켓 크기를 50M개(메모리 400MB)로 고정한 경우(a)와 $|H_2|$는 50M개로, 나머지 버켓 크기는 100K개로 설정한 경우(b)로 구분하여, 펜티엄 기반 리눅스 시스템의 메모리를 256MB, 384MB, 512MB, 640MB, 768M로 변화시키면서 측정한 *DHP*알고리즘의 성능을 그림 3에 보였다. 이 그림에서 시스템 모드 실행시간이 운영체제의 부담에 해당된다. 그림 3의 (a)는 DHP의 메모리 사용량이 커짐에 따라 운영체제의 부담도 증가하지만 프로그램 자체의 수행 가능 여부와는 무관함으로 DHP 알고리즘의 메모리 사용량은 단지 선택의 문제에 지나지 않는다는 사실을 말하며, (b)는 단계 2(H_2)만을 완전 해쉬 테이블로 구현하여 메모리 사용량을 줄였지만 성능은 오히려 크



(a)



(b)

그림 3 시스템 메모리에 따른 DHP의 성능 추이

게 향상되고 있음을 보여주고 있다. 즉, *DHP* 알고리즘의 버켓 크기는 메모리 사용량이 커짐에 따른 전지율 향상 효과와 운영체제 부담 증가 사이의 적절한 균형 문제인 것이다.

결과적으로 그림 2와 그림 3의 내용을 종합하면, VLM 시스템에서 *DHP*의 H_2를 완전 해쉬 테이블로 구현하려는 시도[11]는 메모리 사용량의 최적화를 위한 한 가지 접근 방안으로서 충분한 근거와 타당성을 가진다고 말할 수 있다.

4. 완전 해싱을 위해 *DHP*를 개선한 *PHP* 알고리즘

4.1 *DHP* 알고리즘에서 H_2의 특성

그림 3에서 본 바와 같이, *DHP* 알고리즘에서 H_2는 오직 두 항목에 대한 해쉬 값을 사용하므로 완전 해쉬 테이블로의 구현이 용이하고, 요구되는 메모리 또한 큰 부담이 되지 않는 경우가 많다. 그런데 그림 1의 *DHP* 알고리즘은 H_2가 완전 해쉬 테이블로 구현될 경우 완전 해쉬에 따른 이점을 충분히 반영하지 못한다. 왜냐하면 part-1의 line 6에서 H_2가 완전 해쉬 테이블이므로 모든 가능한 두 항목의 조합에 대한 정확한 지지도가 계수되어 L_2가 결정되는데, part-2에서 이를 다시 계수하고 있기 때문이다. 따라서 H_2가 완전 해쉬 테이블로 구현될 경우 part-2의 line 9에서 단계 2(k=2)는 생략될 수 있어야 한다.

4.2 완전 해싱을 위한 *DHP* 알고리즘의 개선

그림 1의 *DHP* 알고리즘에서 H_2를 완전 해쉬 테이블로 구현하고, part-2의 line 9에서 k=3으로 하여 단순히 단계 2를 건너 뛸 경우, 그림 5에서 보는 바와 같이 단계 3에서 H_3가 없기 때문에 C_3(그림 1의 line 38에서 k=3)와 H_4(그림 1의 line 43에서 k=3)를 구할 수 없고, D_3가 준비되어 있지 않다는 문제에 직면한다.

이를 해결하기 위해 *DHP* 알고리즘을 개선한 *PHP* (Perfect Hash and Pruning) 알고리즘을 그림 4에 제시하였다. *PHP* 알고리즘은 단계 3을 위한 독립된 처리 부분인 part-1′을 도입하여 C_3와 H_4를 다음의 방법으로 생성하고(그림 5 참조), part-2는 단계 4(그림 4의 line 29에서 k=4)부터 시작하도록 하였다.

- C_3를 생성할 때 확장되는 끝 두 항목 i_j, i_k에 대한 $H_2[h_2(i_j, i_k)]$ 값이 최소 지지도를 만족하면 C_3에 포함시킨다(그림 4의 line 46). 예를 들어 $\{a, b\} \in L_2$와 $\{a, c\} \in L_2$로부터 확장된 $\{a, b, c\}$가 빈발 항목 집합이 될 최소한의 조건은 부분 집합인 $\{a, b\}$, $\{a, c\}$, $\{b, c\}$가 모두 빈발 항목 집합이어야 하는데, 앞 두개는 이미 빈발 항목 집합이므로 마지막 $\{b, c\}$ 만이 빈발

```
1 s = a minimum support;                    /* - Part 1 - */
2 set all buckets of H2 to zero;            /*perfect hash*/
3 forall transactions t∈D do begin
4   insert & count 1-items occurrences in hash tree;
5   forall 2-subsets x of t do
6     H2[h2(x)]++;
7 end
8 L1 = {c|c.count≥ s, c is s leaf node of hash tree};
9 L2 = {2-subset x of I | H2[h2(x)] ≥ s};
-------------------------------------------------------
9 k = 3;  D3 = D;                           /* - Part 1' -*/
10 gen_candidate3(L2, H2, C3);
11 set all the buckets of Hk to zero;
12 D4 = ∅;
13 forall transactions t∈D do begin
14   't = ∅; j = 0;
15   forall items ti∈t do
16     if ({ti} ∈ L1) then do begin
17       't j = ti; j++;
18     end
19   if (|'t| ≥ 2) then do begin
20     count_support('t, C3, k, t');  /* t'⊆'t */
21     if (|t'| > 3) then do begin
22       make_hasht4(t', H4); D4 = D4 U {t'};
23     end
24   end
25 end
-------------------------------------------------------
26 k = 4;                                   /* - Part 2 - */
27 while(|{x|Hk[x] ≥ s}| ≥ LARGE) {
28   gen_candidate(Lk-1, Hk, Ck); /* make hash table */
29   set all the buckets of Hk+1 to zero;
30   Dk+1 = ∅;
31   forall transactions t∈D do begin
32     count_support(t, Ck, k, t');  /* t'⊆t */
33     if (|t'| > k) then do begin
34       make_hasht(t', Hk, k, Hk+1, t");
35       if (|t"| > k) then Dk+1 = Dk+1 U {t"};
36     end
37   end
38   Lk = {c∈Ck|count≥ s};
39   k++;
40 }
41                                          /* - Part 3 - */
-------------------------------------------------------
42 Procedure gen_candidate3(L2, H2, C3)
43   C3 = ∅;
44   forall c = cp[1]·cp[2]·cq[2],
45             cp, cq ∈ L2, |cp ∩ cq| = 1  do
46     if (H2[h2({cp[k-1]·cq[k-1]})] ≥ s) then
47       C3 = C3 U{c}; /* insert c into hash tree*/
48 End Procedure
-------------------------------------------------------
49 Procedure make_hasht4(t', H4)
50   forall 4-subsets x(t'1,...,t'4) of t' do
51     H4[h4(x)]++;
52 End Procedure
-------------------------------------------------------
53 /* gen_candidate(), make_hasht()는 그림 1과 동일*/
```

그림 4 완전 해싱을 위해 *DHP*를 개선한 *PHP* 알고리즘

항목 집합이면 된다. 이는 그림 1의 *DHP* 알고리즘에서 H_3를 생성할 때 L_2로부터 C_3로 확장되는 과정에 대한 특수성을 고려하지 않고, 일반화된 부분 집합인 {*a*, *b*}, {*a*, *c*}, {*b*, *c*} 모두를 고려한 것(그림 1의 line 43)보다 효과적이라는 부수적인 이점을 제공하면서도 전지효과는 우수하다.

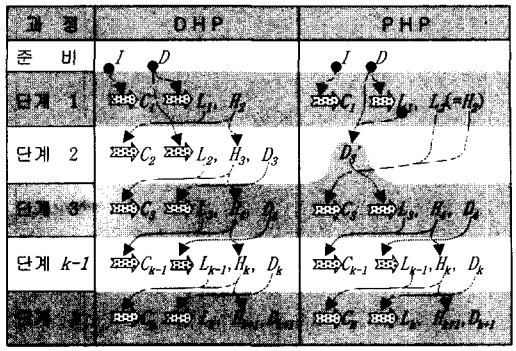


그림 5 L_k, C_k, H_k의 생성과 적용 과정의 비교

- 단계 3을 위한 D_3는 D로 하되(그림 4의 line 9), 각 거래에 대해서 해쉬 트리 검색 및 계수를 실시하기 직전 해당 거래로부터 L_1에 포함되지 못한 항목들을 제거한다(그림 4의 line 15~18). 이는 *DHP* 알고리즘에서 H_2를 고려하여 H_3를 계수하면서 전지한 D_3(그림 1의 line 17~18)보다는 우수하지 못하다.
- H_4는 단순히 현재 거래에 포함된 모든 4-itemset 부분 집합에 대하여 계수하고, 이와 같은 부분 집합이 하나라도 있는 거래는 D_4에 포함시킨다(그림 4의 line 21~22). 따라서 H_4와 D_4는 *DHP* 알고리즘에 비해서 우수하지 못하다.

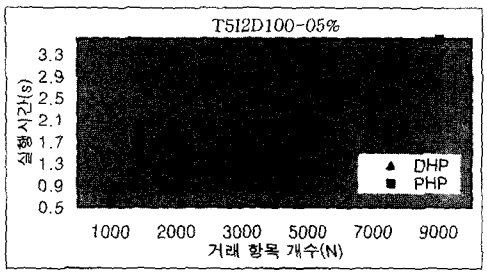
4.3 *DHP*와 *PHP* 알고리즘의 성능 비교

[3]에 의해서 생성된 다양한 유형의 거래 데이타에 지지도 0.5%와 0.75%를 적용하여 측정한 *DHP*와 *PHP* 알고리즘의 성능 결과를 표 5에 요약하고, 그 추이를 그림 6에 나타내었다. 이 표와 그림에 의하면 지지도에 관계 없이 거의 모든 데이타 영역에서 *PHP*가 우수하고, T20I2,N1000~T20I6,N1000과 T5I2,N5000~T10I4,N9000 등의 일부 영역에서 *DHP*가 우수하게 나타나고 있는데, 그 이유는 다음과 같이 유추할 수 있다.

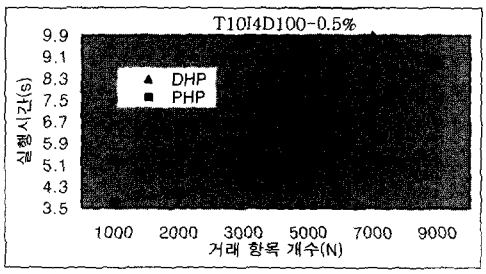
표 5 *DHP*와 *PHP*의 성능 비교(0.5%, 0.75%)

DB \ N	1000	2000	3000	5000	7000	9000
T5I2	▲	▲	▲	▽	▽	△
T10I2	▲	▲	▲	▲	▲	▽
T10I4	▲	▲	▲	▲	▲	▽
T15I4	▲	▲	▲	▲	▲	▲
T20I2	△	▲	▲	▲	▲	▲
T20I4	△	▲	▲	▲	▲	▲
T20I6	△	▲	▲	▲	▲	▲

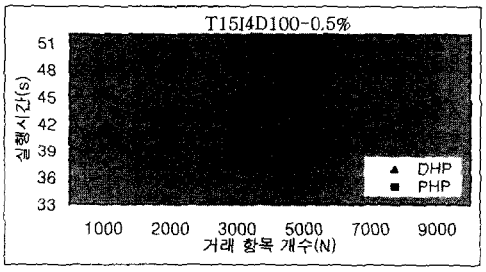
▲ *PHP* 우수 ▽ 0.75%에서만 *DHP* 우수 △ *DHP* 우수



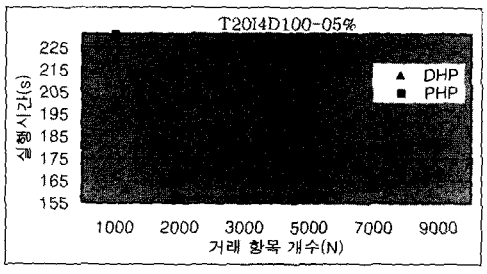
(a)



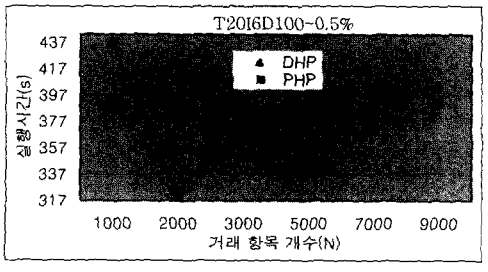
(b)



(c)



(d)



(e)

그림 6 표 5에 따른 DHP와 PHP의 성능 비교

- 표 5의 왼쪽 아래 부분은 거래에 포함된 항목의 분포가 밀집되어 있으므로 단계 4까지 빈발 항목 집합이 매우 많은 영역인데, *PHP* 알고리즘에서 단계 2가 생략됨으로써 얻어지는 이득보다는 'H_3의 부재→H_4의 효율성 저하→C_4의 전지율 하락→C_4 검색 및 계수 부담 증가'에 의한 비용이 더 크기 때문에 *DHP*가 우수하다.
- 반대로 표 5의 오른쪽 위 부분은 거래 항목이 매우 희박하게 분포하면서 빈발 항목은 상대적으로 적은 영역으로, 완전 해쉬 테이블로 얻어지는 이득보다는, 많은 메모리가 필요한 해쉬 테이블 자체(sizeof(int) *9000* 8999/2≈약 160MB)의 초기화를 위한 부담이 더 크기 때문에 *PHP*가 불리하다.

*PHP*가 우수한 영역에서 *PHP*의 *DHP* 대비 성능 향상 비율은 최대 43%, 평균 18.7%로 나타났고, 지지도 0.75%와 0.5%를 합친 영역별 평균 성능 향상 추이는 그림 7과 같다. 이 그림으로부터 거래 및 빈발 항목 집합의 평균 크기가 큰 T20I6과 T20I4 유형에서의 성능 향상이 상대적으로 낮음을 알 수 있는데, 그 이유는 앞서 언급한 바와 같이 C_4의 전지율 하락에 따라 *PHP*의 효과가 반감되기 때문이다.

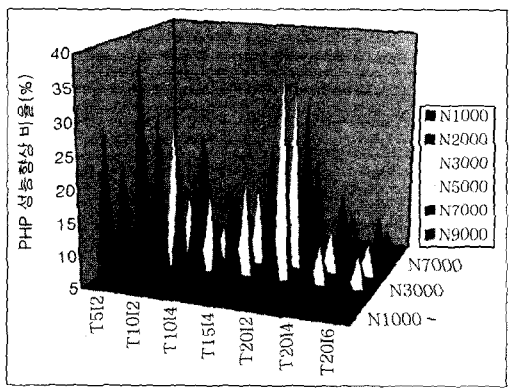


그림 7 데이타 영역별 PHP 성능 향상 추이

5. 결 론

전체 항목 수가 큰 경우 H2를 완전 해쉬 테이블로 구현할 경우 메모리 요구량이 증가하여 완전 해싱으로부터 얻어지는 이득보다는 해쉬 테이블 초기화 및 운영 체제의 부담이 오히려 더 커질 우려가 있다. 이런 현상이 물리 메모리가 적은 시스템에서는 두드러지게 나타나지만, GB 단위의 VLM 시스템에서는 크게 완화되어 큰 문제가 되지 않음을 다양한 실험을 통해 살펴보았다.

그러나 단순히 H_2를 완전 해쉬 테이블로 구현한 *DHP* 알고리즘은 완전 해싱의 이점을 충분히 활용하지

못하기 때문에, 이를 보완하기 위해서 *DHP*의 단계 2를 건너뛸 수 있도록 개선한 *PHP* 알고리즘을 제안하고, 그 우수성을 다양한 시험용 데이타를 대상으로 검증하였다. 그 결과 극히 일부 영역을 제외한 거의 모든 영역에서 지지도와 관계 없이 최대 43%, 평균 18.7%의 성능 향상 효과가 있음을 보였다.

여기서 제안된 *PHP* 알고리즘은 최근 보편화되고 있는 VLM 시스템에 적용될 수 있는 현실적인 방안이라 점에서 의의가 크다고 본다. 여기에, C_3를 계수하기 위한 $D_3(=D)$의 사전 전지와 C_4의 전지를 위한 H_4의 정확성 향상을 위한 연구가 계속 진행되고 있으므로 *PHP* 알고리즘의 효용성은 더욱 높아질 것이다.

참 고 문 헌

[1] R. Agrawal and R. Srikant, *"Fast Algorithms for Mining Association Rules,"* Proceedings of the 20th International Conference on Very Large Databases, pp.487-499, 1994.

[2] J. S. Park, M.-S. Chen and P. S. Yu, *"An Effective Hash-Based Algorithm for Mining Association Rules,"* Proceedings of ACM SIGMOD, pp.175-186, 1995.

[3] R. Agrawal and et al, *"Synthetic Data Generation Code for Associations and Sequential Patterns,"* http://www.almaden.ibm.com/cs/quest, 1999.

[4] R. Agrawal, T. Imielinski and A. Swami, *"Mining Association Rules between Sets of Items in Large Databases,"* Proceedings of ACM SIGMOD on Management of Data, pp.207~216, 1993.

[5] M. Houtsma, and A. Swami, *"Set-oriented mining of association rules,"* Proceedings of the International Conference on Data Engineering, pp.26~33, 1995.

[6] A. Savasere, E. Omiecinski and S. Navathe, *"An Efficient Algorithm for Mining Association Rules in Large Databases,"* Proceedings of the 21th VLDB Conference, pp.432~444, 1995.

[7] 이재문, 박종수, "복합 해쉬 트리를 이용한 효율적인 연관 규칙 탐사 알고리즘", 정보과학회 논문지(B) 제 26권, 제 3호, pp.343~352, 1999.

[8] Digital SPD, *"Very Large Memory Applications,"* http://wint.decsy.ru/du/dec_unix/64bit/vlm.htm.

[9] D. Irwin, *"Oracle Very Large Memory(VLM) for Digital Alpha NT,"* Oracle white paper, 1997.

[10] U. Vahalia, *UNIX Internals, The New Frontier,* Prentice Hall, p.400, 1996.

[11] 이재문, "대용량 주기억장치 시스템에서 효율적인 연관 규칙 탐사 알고리즘", 정보처리학회 논문지D 제9-D권, 제4호, pp.579-586, 2002.

이 형 봉

1984년 서울대학교 계산통계학과 이학사 1986년 서울대학교 대학원 계산통계학과(전산과학) 이학석사. 2002년 강원대학교 대학원 컴퓨터과학과 이학박사. 1986년~1994년 LG전자 컴퓨터 연구소 선임연구원. 1994년~1999년 한국디지탈이큅먼트㈜ 책임. 1997년~1999년 전자계산조직응용, 전자계산기, 정보통신 기술사. 1999년~2003년 호남대학교 정보통신공학부 조교수. 2004년~현재 강릉대학교 컴퓨터공학과 조교수. 관심분야는 프로그램 언어 및 보안, 운영체제, 알고리즘, 멀티미디어 통신