논문개요

최근 정보 산업 분야에서 데이터의 양적 팽창과 그러한 데이터를 유용한 정보와 지식으로 바꿔야 하는 필요성으로 인해 데이터 마이닝에 대한 연구가 활발히 이루어지고 있다. 이러한 데이터 마이닝 분야에서 새롭게 연구되고 있는 한 분야가 순회 패턴 탐사이다.

본 논문에서는 순차 패턴 탐사의 한 특별한 경우인 순회 패턴 탐사 문제에 대해 연구하였다. 객체들은 서로 연결되어 상호 접근이 허용되며 객체 사이의 접근은 서로 연결된 경로를 따라 일정한 방향성을 갖는다. 이러한 환경하에서 객체들을 순차적으로 접근하는 일정한 패턴을 발견하고자 한다. 예를 들어, 버스나 지하철을 이용한 승객들의 이동 경로를 분석하여 일정한 패턴을 발견할 수 있다. 이러한 문제를 해결하기 위해 기존의 알고리즘들을 적용하여 해결하고자 했다. 즉, 순차 패턴 탐사 알고리즘인 GSP와 SPADE를 변형하여 적용한 GSP_V와 SPADE_V를 구현하였으며, 순회 패턴 탐사 알고리즘인 FS를 변형하여 적용한 FS_V를 구현하였다. 그리고 빠른 탐사를 위해 기존에 존재하는 알고리즘의 일부 특성에 다양한 자료구조를 복합 적용한 AVL(Array and Vertical List)를 구현하였다. 이렇게 구현한 알고리즘들에 대해 실세계 데이터인 대용량의 교통 카드 트랜잭션 데이터베이스를 적용시켜 보았고, 최소 지지도나 입력 데이터베이스 크기 등에 따른 각 알고리즘의 성능을 비교 분석해 보았다.

목　차

논문개요

Ⅰ. 서론 ………………………………………………………………………………………… 1

Ⅱ. 순회 패턴 탐사 ……………………………………………………………………… 4
1. 순회 패턴 탐사 정의 …………………………………………………………… 4
2. 관련 연구 ……………………………………………………………………………… 9
1) FS 알고리즘 …………………………………………………………………… 10
2) CHT_FS 알고리즘 …………………………………………………………… 11
3) TPADE 알고리즘 …………………………………………………………… 11

Ⅲ. 순회 패턴 탐사 알고리즘 …………………………………………………… 13
1. SPADE_V 알고리즘 ………………………………………………………… 13
1) 빈발 1-시퀀스(F_1) ……………………………………………………… 17
2) 빈발 2-시퀀스(F_2) ……………………………………………………… 17
3) 빈발 k-시퀀스(F_k) ……………………………………………………… 18
① 후보 k-시퀀스(C_k) 생성 …………………………………………… 18
② 빈발 k-시퀀스(F_k) 발견 …………………………………………… 19
2. GSP_V 알고리즘 ……………………………………………………………… 21
1) 빈발 1-시퀀스(F_1) ……………………………………………………… 22
2) 빈발 2-시퀀스(F_2) ……………………………………………………… 23
3) 빈발 k-시퀀스(F_k) ……………………………………………………… 23
3. FS_V 알고리즘 ………………………………………………………………… 24
1) 빈발 1-시퀀스(F_1) ……………………………………………………… 26

2) 빈발 2-시퀀스(F_2) ······ 26
3) 빈발 k-시퀀스(F_k) ······ 27
4. AVL 알고리즘 ······ 28
1) 빈발 1-시퀀스(F_1) ······ 30
2) 빈발 2-시퀀스(F_2) ······ 32
① 후보 2-시퀀스(C_2) 생성 ······ 32
② 빈발 2-시퀀스(F_2) 발견 ······ 35

Ⅳ. 실험 결과 및 분석 ······ 37
1. 실험 환경 및 실험 데이터 특성 ······ 37
2. 시험 결과 및 성능 비교 분석 ······ 40
1) 실험 결과 ······ 40
① 시퀀스 길이에 따른 후보 시퀀스와 빈발 시퀀스 ······ 40
② 최소 지지도에 따른 탐사된 최대 빈발 시퀀스 ······ 42
2) 성능 비교 분석 ······ 43
① 입력 데이터 수에 따른 수행 시간 ······ 44
② 시퀀스 길이에 따른 각 알고리즘의 실행 시간 ······ 47
③ 최소 지지도에 따른 각 알고리즘의 실행 시간 ······ 48
④ FS_V 알고리즘의 비트연산 vs. 덧셈 연산 ······ 51

Ⅴ. 결론 ······ 53

참고문헌
ABSTRACT

그림 목차

그림 2.1 : 교통 노선도 ······ 7

그림 2.2 : 트랜잭션 데이터베이스 ······ 7

그림 2.3 : 빈발 시퀀스 ······ 8

그림 2.4 : 최대 빈발 시퀀스 ······ 8

그림 3.1 : SPADE 알고리즘 ······ 14

그림 3.2 : 수직 데이터베이스 예제 ······ 16

그림 3.3 : 빈발 2-시퀀스의 정렬 합병 조인 ······ 19

그림 3.4 : 식별자 리스트 접합 과정 ······ 20

그림 3.5 : GSP 알고리즘 ······ 21

그림 3.6 : AVL 알고리즘 ······ 29

그림 3.7 : F_1에서의 해시 테이블 ······ 31

그림 3.8 : C_2 생성에서의 lookup 해시 테이블 ······ 34

그림 3.9 : C_2 생성에서의 이차원 배열 ······ 34

그림 4.1 : 전 처리된 트랜잭션 데이터베이스 ······ 39

그림 4.2 : 최소 지지도에 따른 탐사된 최대 빈발 시퀀스 ······ 42

그림 4.3 : 입력 데이터 수에 따른 각 알고리즘의 실행 시간 ······ 46

그림 4.4 : 시퀀스 길이에 따른 각 알고리즘의 실행 시간 ······ 48

그림 4.5 : 최소 지지도에 따른 각 알고리즘의 실행 시간 ······ 50

그림 4.6 : 시퀀스 길이에 따른 각 최소 지지도별 탐사된 빈발 시퀀스 ··· 50

표 목차

표 4.1 : 시퀀스 길이에 따른 후보 시퀀스와 빈발 시퀀스의 개수 ················ 41

표 4.2 : 탐사된 최대 빈발 시퀀스 예제 ·· 43

표 4.3 : FS_V 알고리즘의 시퀀스 길이에 따른 데이터베이스 크기 ············ 52

표 4.4 : FS_V 알고리즘의 시퀀스 길에 따른 실행 시간 ······························· 52

Ⅰ. 서론

최근 대규모 데이터 집합 속에 숨겨진 흥미 있는 데이터 패턴을 찾기 위한 데이터 마이닝 기법들이 폭넓게 연구되고 있다. 그러한 데이터 마이닝 기법 중에 하나가 연관 규칙[11, 8, 12, 13]이다. 연관 규칙은 대규모 데이터 항목 집합 사이에서 유용한 연관성과 상관관계를 찾는 기법이다. 지속적으로 많은 데이터들이 수집되고 저장되어 오는 동안 많은 산업분야에서는 데이터베이스 내의 연관 규칙을 발견하는 것이 유용하다고 인식하게 되었다. 연관규칙 탐사의 전형적인 활용 중에 하나가 장바구니 분석이다. 이 프로세스는 고객들의 장바구니에서 서로 다른 품목들 사이의 연관관계를 발견함으로써 고객의 구매습관을 분석한다. 이는 고객들이 빈번하게 함께 구매한 품목들에 대한 직관을 갖게 함으로써 기업이나 소매상들이 마케팅 전략을 세우는데 많은 도움을 준다. 이러한 연관 규칙 탐사의 응용 중에 하나가 순차 패턴 탐사[1, 2, 3, 6, 7, 9]이다. 순차 패턴 탐사는 한 트랜잭션 안에서 발생하는 항목들 간의 연관 규칙에 시간의 변이를 추가한 것이다[7]. 즉 연관 규칙은 트랜잭션 안에서 어떤 항목을 함께 구매하는가에 대한 문제로 트랜잭션 내의 문제인 반면, 순차 패턴을 발견하는 것은 트랜잭션 상호간의 문제인 것이다[7]. 연관 규칙의 한 응용이자 순차 패턴의 한 특별한 형태인 순회 패턴 탐사는 최근 웹 비즈니스가 급증하면서 그 연구가 활발히 진행되고 있다[16, 14, 5, 15]. 웹을 이용하는 사용자의 관심과 행동 양식에 관한 정보의 요구가 높아지면서 웹에서 발생하는 데이터들에 데이터 마이닝 기법을 적용하여 유용한 패턴 정보들을 찾아내고자 하는 연구가 이루어지고 있다. 그 대표적인 예가 웹 로그 파일에서

일반 사용자가 접근한 웹 페이지를 분석하여 순차적으로 접근하는 일정한 패턴을 찾아내는 순회 패턴 탐사이다. 이러한 웹 환경 하에서의 순회 패턴 탐사 알고리즘에 대한 연구는 많은 반면 그 외의 응용에서 순회 패턴 탐사에 대한 연구는 그리 많지 않다.

이에 본 논문에서는 교통 카드에 의한 트랜잭션 데이터베이스에서 승객들의 이동 경로인 순회 패턴을 탐사하는 알고리즘에 대해 연구하였다. 트랜잭션 데이터베이스는 각 레코드가 하나의 트랜잭션을 나타내는 파일로 구성되어 있다. 교통 카드에 의한 트랜잭션 데이터베이스는 교통 카드를 사용한 승객들의 트랜잭션들로 구성되어 있고, 교통 카드를 사용한 승객의 한 트랜잭션은 승객 식별자와 트랜잭션 식별자 그리고 그 트랜잭션을 구성하는 항목 리스트로 구성되어 있다. 항목 리스트는 그 승객이 승차하여 순차적으로 경유한 정류장들에 관한 자료를 가지고 있다. 이와 관련하여 각 트랜잭션에는 추가적인 정보들을 포함하고 있으나 순차 패턴 탐사에 필요한 정보들만 갖도록 선 처리 되어졌다. 이러한 입력 데이터베이스에서 어떻게 하면 효율적으로 순회 패턴을 탐사할 수 있는지에 대해 그 방법을 연구하였다. 효율적인 순회 패턴 탐사를 위해 기존의 순회 패턴 탐사나 순차 패턴 탐사 또는 연관 규칙 탐사 알고리즘들에서 사용하였던 다양한 알고리즘을 접목시켜 보았다. 기존의 알고리즘을 위의 순회 패턴 탐사 특성에 맞게 적용시켜 구현한 알고리즘들은 다음과 같다. SPADE 알고리즘을 적용하여 교통 카드 트랜잭션 데이터베이스의 특성에 맞게 변형시킨 SPADE_V를 구현하였고, GSP 알고리즘을 적용하여 GSP_V를 구현하였으며, FS 알고리즘을 적용하여 FS_V를 구현하였다. 마지막으로 배열과 SPADE 알고리즘에서 제안한 수직 데이터베이스 개념을 복합 적용한 AVL(Array and Vertical List)을 구현하였다. 위의 각 알고리즘이 어떠한 성능을 보이는지 교통 카드 트랜잭션 데이터베이스를 적용시켜 실험 해 보았

고, 그 결과를 비교 분석해 보았다.

본 논문의 구성은 다음과 같다. 제 Ⅱ장에서는 순회 패턴 탐사에 대해 그 문제 정의와 관련 연구에 대하여 살펴보고, 제 Ⅲ장에서는 주어진 문제에서 패턴을 찾는 순회 패턴 탐사에 대한 각 알고리즘을 설명한다. 제 Ⅳ장에서는 구현된 각 알고리즘의 실험결과에 대해 알아보고 마지막으로 제 Ⅴ장은 결론으로 끝을 맺는다.

Ⅱ. 순회 패턴 탐사

1. 순회 패턴 탐사 정의

트랜잭션 데이터베이스에서의 순회 패턴 탐사 문제는 다음과 같이 정의할 수 있다. 객체(Object)들은 서로 연결되어 있으며, 객체들 사이의 접근은 서로 연결된 경로를 따라 일정한 방향성을 가지며 접근한다. 이러한 환경 하에서 객체들을 순차적으로 접근하는 일정한 패턴을 발견하는 것을 순회 패턴 탐사라고 한다.

위와 같이 정의한 순회 패턴 탐사 문제는 순차 패턴 탐사의 한 특별한 경우로써, 웹 환경 하에서의 순회 패턴 탐사 문제를 다룬 FS[4]와 그 차이가 있다. FS의 경우 객체(웹 페이지)의 접근에 있어 역방향 접근의 문제를 다루는 반면, 본 논문에서 정의한 순회 패턴 탐사의 경우 객체를 접근하는데 일정한 방향성을 유지한다. 순회 패턴을 탐사하는 문제는 다음과 같이 표현될 수 있다. I = {i_1, i_2, ..., i_m}는 서로 다른 m개의 항목들의 집합이다. 한 시퀀스(*sequence*)는 항목들의 순서 유지 리스트이다. 한 시퀀스 α는 ($\alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_n$)로 나타내고, α_i는 한 항목이다. 기존의 순차 패턴 탐사의 경우 α_i는 항목 집합이 올 수 있었지만, 순회 패턴 탐사의 경우 반드시 하나의 항목만이 올 수 있다. *k*개의 항목을 가진 한 시퀀스를 *k*-시퀀스라고 부른다. 예를 들어, (2 → 1 → 3)는 3-시퀀스이다. 모든 *i*에 대해, $1 \leq i \leq n$, $1 \leq j + i - 1 \leq m$ 그리고 $n \leq m$을 만족하는 경우, 한 시퀀스 $\alpha = (\alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_i \rightarrow \cdots \rightarrow \alpha_n)$는 다른 시퀀스 $\beta = (\beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \beta_j \rightarrow \cdots \rightarrow$

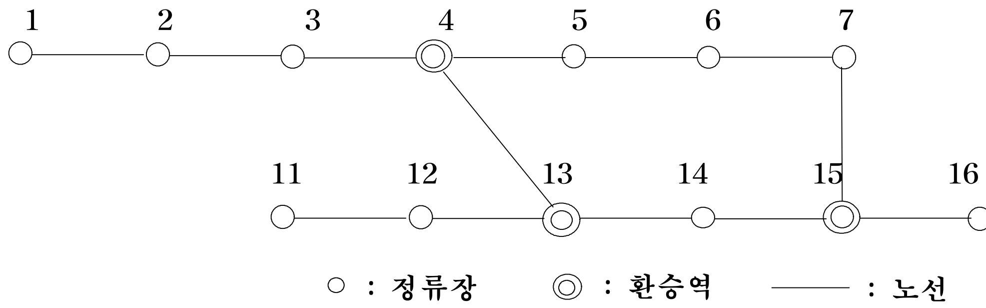
β_m)의 부분시퀀스이고 $\alpha \leq \beta$로 표기 한다. 예를 들어, (4 → 5)는 (4 → 5 → 6)의 부분시퀀스이다. 반면, (4 → 6)은 (4 → 5 → 6)의 부분시퀀스가 아니다. 기존의 순차 패턴의 경우 이러한 시퀀스도 부분시퀀스에 포함되지만 순회 패턴 탐사의 경우는 항목의 순서 유지뿐만 아니라 연속적으로 발생해야 하기 때문에 부분시퀀스가 될 수 없다.

만약 $\alpha \leq S$ 인 경우, 한 시퀀스 S는 다른 시퀀스 α를 포함한다고 한다. 즉 α는 S의 부분시퀀스이다. 한 시퀀스의 지지도나 빈발은 데이터베이스에서 그 시퀀스를 포함하는 트랜잭션의 총 개수이다. 최소 지지도(*min_sup*)라고 불리는 사용자가 정의한 임계값이 주어졌을 때, 한 시퀀스가 *min_sup*보다 많이 발생한 경우를 빈발(*frequent*)하다고 한다. 빈발 k-시퀀스의 집합을 F_k로 표기하고, 후보 k-시퀀스의 집합을 C_k로 표기한다. 한 빈발 시퀀스가 다른 시퀀스의 부분시퀀스가 아닐 경우 최대(*maximal*)라고 한다.

이 부분 이하에서의 순회 패턴 탐사 문제는 위에서 정의한 문제를 해결하기 위한 것을 일컫는다. 순회 패턴 탐사를 위한 트랜잭션 데이터베이스는 각 레코드가 하나의 트랜잭션을 나타내는 파일로 구성된다. 그 데이터베이스에서의 각 트랜잭션은 CID라 불리는 고객 식별자와 TID라 불리는 트랜잭션 식별자 그리고 항목 리스트를 갖는다. 트랜잭션 데이터베이스는 CID를 주키로 TID를 부키로 정렬되어 있다. 항목들의 리스트는 시간 변이에 따른 리스트이고 한 타임스탬프에 하나의 항목만 올 수 있다. 그리고 트랜잭션은 부가적인 몇 가지 정보를 추가로 포함할 수 있다. 한 트랜잭션에서 항목들의 부분시퀀스는 중복해서 발생할 수 있다. 한 트랜잭션에서 중복 발생한 부분 시퀀스에 대한 지지도 계산은 중복 발생한 부분 시퀀스가 그 트랜잭션에서 여러 번 발생해도 지지도 계산에는 한번만 참여한다. 예를 들어, 빈발 1-시퀀스 지지도 계산을 고려해 보자. 그림 2.2의 마지막 줄에 해당하는 트랜잭션의 항목들 중

4는 그 트랜잭션에서 두 번 발생하지만, 그 지지도 계산에는 한번만 참여한다. 그래서 전체 트랜잭션 데이터베이스에서 4는 그 지지도가 8인 것이다.

트랜잭션 데이터베이스 *D*와 *min_sup*가 주어졌을 때, 순회 패턴을 탐사하는 문제는 그 트랜잭션 데이터베이스에서 모든 최대 빈발 시퀀스를 발견하는 것이다. 그림 2.2의 교통 카드 트랜잭션 데이터베이스를 고려해 보자. 이 트랜잭션 데이터베이스는 그림 2.1의 교통 노선도를 이용한 승객들의 이동 경로에 대한 데이터베이스이다. 그 한 예로써, 2번 승객의 2번 트랜잭션은 두 번 환승하였으며, 그 이동 경로는 정류장 1번에서 승차해서 2, 3, 4, 13을 경유하여 12번 정류장에서 하차하였다. 원래의 교통카드 트랜잭션 데이터베이스는 그림 2.2의 예제 데이터베이스보다 훨씬 더 많은 정보를 포함하고 있으며, 그 이동 경로인 항목 리스트는 포함되지 않는다. 이는 승차역과 하차역 사이의 경로를 채워 넣는 작업으로 사전 처리 되어졌다. 이에 대한 자세한 사항은 [18]을 참조하다. 이 트랜잭션 데이터베이스는 1부터 7까지 그리고 11부터 16까지 총 13개의 항목들로 구성되고, 각 트랜잭션은 CID와 TID의 조합으로 식별할 수 있다. 그림 2.2의 트랜잭션 데이터베이스와 최소 지지도 40%가 주어졌을 때, 순회 패턴 탐사 문제는 그림 2.4와 같이 최대 빈발참조 시퀀스를 발견해내는 것이다. 그림 2.3은 순회 패턴 탐사의 중간 과정으로써 최소 지지도 40%를 만족하는 모든 빈발 *k*-시퀀스를 나타낸 것이다. 본 논문에서는 전반적으로 이 예제를 실행 예제로 사용할 것이다.



[그림 2.1] 교통 노선도

최소 지지도: 40%

CID	TID	환승회수	트랜잭션 내의 항목 개수	트랜잭션 내의 항목 리스트
1	1	0	6	1 2 3 4 5 6
1	2	2	5	3 4 13 14 15
1	3	0	6	3 4 5 6 7 15
1	4	0	3	4 5 6
2	1	1	6	7 15 14 13 12 11
2	2	2	6	1 2 3 4 13 12
3	1	0	5	4 5 6 7 15
4	1	1	5	12 13 14 15 7
4	2	0	5	3 4 5 6 7
4	3	2	9	3 4 5 6 7 15 14 13 4

[그림 2.2] 트랜잭션 데이터베이스

빈발 1-시퀀스	지지도
3	6
4	8
5	6
6	6
7	6
13	5
14	4
15	6

빈발 2-시퀀스	지지도
3 → 4	6
4 → 5	6
5 → 6	6
6 → 7	4
7 → 15	4

빈발 3-시퀀스	지지도
3 → 4 → 5	4
4 → 5 → 6	6
5 → 6 → 7	4

빈발 4-시퀀스	지지도
3 → 4 → 5 → 6	4
4 → 5 → 6 → 7	4

[그림 2.3] 빈발 시퀀스

최대 빈발 시퀀스	지지도
7 → 15	4
3 → 4 → 5 → 6	4
4 → 5 → 6 → 7	4

[그림 2.4] 최대 빈발 시퀀스

2. 관련 연구

순회 패턴 탐사에 대한 연구는 대부분 웹 환경에서 사용자의 웹페이지 접근 패턴을 탐사하는 것에 대한 연구가 많이 이루어져 왔다[16, 5, 14, 15]. 그리고 [4]에서는 연관 규칙 탐사와 순차 패턴 탐사 알고리즘들을 응용한 순회 패턴 탐사를 연구하였고 다음과 같은 특징을 갖는다.

[4]는 순회 패턴 탐사 문제를 다음과 같이 형식화하였다. 객체들(Objects)이 서로 연결되어 있는 정보 제공 환경 하에서 사용자는 제공된 링크와 아이콘에 따라 앞뒤로 객체를 순회하기 쉽다. 그 결과 어떤 노드는 그 내용보다 그 위치 때문에 방문하게 될 가능성이 많다. 예를 들어, WWW 환경 하에서 형제 노드를 방문하기 위해서 새로운 URL을 여는 대신 역방향 아이콘을 누르고 다시 순방향으로 형제 노드를 선택한다. 결과적으로 원래 로그 데이터베이스에서 의미 있는 사용자 접근 패턴을 추출하기 위해서는 이러한 역방향 순회의 영향을 고려하여 관심 있는 실제 접근 패턴을 발견하도록 한다. 역방향 참조가 일어나면 순방향 참조는 종결되고 이렇게 얻어지는 순방향 참조 경로를 최대 순방향 참조(max*i*mal *f*orward re*f*erence)라 한다. 최대 순방향 참조를 구하는 자세한 내용은 [4]를 참조한다. 이러한 최대 순방향 참조를 기반으로 연관 규칙 탐사와 순차 패턴 탐사 알고리즘들을 응용하여 제안된 순회 패턴 탐사 알고리즘은 다음과 같이 3 단계로 구성된다.

단계 1: 원 로그 데이터베이스에서 최대 순방향 참조들을 구한다.

단계 2: 최대 순방향 참조 집합에서 빈발 참조 시퀀스를 결정한다.

단계 3: 빈발 참조 시퀀스에서 최대 참조 시퀀스를 찾는다.

위의 단계 2에서 FS 알고리즘[4]이 적용되고, 이 부분의 성능 개선을 보이는 알고리즘이 CHT_FS와 TPADE 알고리즘[5]이다. 즉, 단계1과 단계3은 세 알고리즘이 모두 같고 단계2 부분만 다르다. 다음은 각 알고리즘의 특징을 알아본다.

1) FS 알고리즘

먼저 FS(Full Scan) 알고리즘[4]는 1단계에서 구해진 최대 순방향 참조들을 포함하는 데이터베이스 D_F에서 빈발하게 발생하는 참조 시퀀스들을 발견함으로써 빈발 순회 패턴을 얻어낼 수 있다. 연관 규칙 탐사 알고리즘인 DHP(standing *f*or direct hashing and pruning) 알고리즘[8]의 개념을 이용한 FS 알고리즘은 빈발 참조 시퀀스를 구한다. DHP 알고리즘은 연관 규칙을 찾는데 있어 두 가지 중요한 특성을 갖는다. 하나는 해시 기법을 이용함으로 빈발 항목 집합의 효과적인 생성을 들 수 있고(특히, 빈발 2-항목 집합 생성에 효과적), 다른 하나는 데이터베이스를 읽은 후 트랜잭션 데이터베이스 크기를 효과적으로 줄이는 전지(Pruning)기법을 들 수 있다.

FS 알고리즘에서 L_k는 모든 빈발 k-참조 시퀀스들의 집합을 나타내고, C_k는 후보 k-참조 시퀀스들의 집합을 나타낸다. D_F를 스캔해서 L_1을 얻고, 각 2-참조 시퀀스의 발생 빈도를 세기위해 해시테이블(H_2)을 만든다. k = 2일 때부터 시작하여 FS는 이전 단계에서 얻어진 해시테이블을 기반으로 하여 C_k를 생성하고, 빈발 k-참조를 결정하고, 다음 단계를 위해 데이터베이스 사이즈를 줄이고, 후보 (k + 1)-참조를 만들기 위해 해시테이블을 만

든다. 여기에서 중요한 점은 연관 규칙에서처럼 C_k를 생성할 때 L_{k-1}을 self-join으로 생성하는 것이 아니라는 점에서 큰 차이점이 있다. 즉 FS에서는 한 시퀀스의 첫 번째 원소와 다른 시퀀스의 마지막 원소를 제거한 후 남아 있는 두 (k-2)-참조 시퀀스가 동일한 경우에만 k-참조 시퀀스를 만들기 위해 조인한다. 그리고 FS는 C_k를 찾기 위해 정돈된(trimmed) 데이터베이스를 읽는다.

2) CHT_FS 알고리즘

CHT_FS(Compound Hash Tree_Full Scan) 알고리즘은 FS 알고리즘에 복합 해시트리를 적용한다[5]. 복합 해시 트리는 FS 알고리즘에서 사용하는 일반적인 해시 트리의 확장이다. 즉, 일반적으로 검색되어야 할 레코드가 외부 노드에만 저장되는 반면 복합 해시 트리는 검색되어야 할 레코드가 내부 노드에도 저장될 수 있다 따라서 후보 항목 집합 C_m, C_{m+1}, ..., C_{m+n}을 동시에 생성한 후 이들에 대하여 하나의 복합 해시 트리를 생성함으로써 해시 트리에서의 탐색시간과 데이터베이스 스캔 비용을 줄이게 된다. CHT_FS 알고리즘의 성능은 FS 알고리즘보다 조금 더 향상되었다[5].

3) TPADE 알고리즘

TPADE(Traversal Patterns Discovery using E*q*uivalence Classes) 알고리즘은 순차 패턴 탐사 알고리즘의 하나인 SPADE 알고리즘[1]을 응용하여 웹 환경 하에서 사용자가 접근한 웹 페이지의 접근 패턴을 탐사할 수

있게 변형시킨 알고리즘이다[5]. TPADE 알고리즘은 SPADE 알고리즘의 특징을 이용하면서 순회 패턴 탐사에 맞게 동치 클래스 분해 부분을 변형하였다. 즉, 주어진 빈발 $(k-1)$참조 시퀀스 집합, F_{k-1}에서 두 개의 시퀀스 중 한 시퀀스의 첫 번째 항목과 나머지 다른 시퀀스의 마지막 항목을 떨어뜨린 후, 남아 있는 부분들 즉, $k-2$ 길이의 중위(*infix*)가 같은 경우 두 시퀀스를 조인하였다. 조인 과정에서 두 시퀀스의 트랜잭션 식별자를 비교하여 같은 트랜잭션에 있는 경우 식별자 리스트에 저장되어 있는 위치 정보를 이용하여 연속적인지를 확인하고, 연속적이면 새로운 후보 시퀀스의 식별자 리스트에 트랜잭션 내에서 앞쪽에 위치한 시퀀스의 식별자 리스트를 추가한다. 즉 제한된 후보 시퀀스를 생성하는 부분에서 SPADE 알고리즘과 다르다. 자세한 내용은 [5]를 참조한다. TPADE 알고리즘의 수행 결과는 기존의 FS 알고리즘과 CHT_FS 알고리즘에 비해 우수하다[5].

Ⅲ. 순회 패턴 탐사 알고리즘

1. SPADE_V 알고리즘

SPADE(Se*q*uential Pattern Discovery using E*q*uivalence Classes) 알고리즘은 순차 패턴을 빠르게 찾기 위한 알고리즘 중에 하나이다. 순차 패턴을 찾기 위한 기존의 알고리즘들은 반복적으로 데이터베이스를 스캔하고, 집약성이 적은 복잡한 해시 구조를 사용한다. 반면, SPADE 알고리즘은 문제를 좀 더 작은 부분으로 나누어 효과적인 래티스(Lattice) 탐색 기법을 사용하고, 간단한 결합 연산들을 사용하여 메인 메모리에서 독립적으로 해결할 수 있는 결합 원리를 이용한다. 단지 세 번의 데이터베이스 스캔으로 모든 시퀀스들이 탐사된다.

SPADE 알고리즘의 주요 특성은 다음과 같다. 첫째, 타임스탬프와 함께 각 시퀀스가 발생하는 것에 대한 객체들의 한 리스트를 연관시키는 수직 식별자 리스트 데이터베이스 형태를 사용한다. 모든 빈발 시퀀스는 간단한 식별자 리스트 결합을 통해 찾아낼 수 있다. 둘째, 탐사 공간을 메인 메모리상에서 독립적으로 수행할 수 있도록 작은 부분 문제들로 나누기 위한 래티스 이론 방식을 사용한다. SPADE 알고리즘은 일반적으로 세 번의 데이터베이스 스캔만을 필요하고, 전 처리된 정보가 있을 경우 오직 한 번의 스캔만이 필요하다. 셋째, 패턴 탐사로부터 문제를 나누는 과정을 분리한다. 각 부분 래티스 내의 빈발 시퀀스를 찾기 위해 넓이 우선 탐사와 깊이 우선 탐사의

두 가지 다른 탐사 방법을 사용한다.

SPADE 알고리즘은 데이터베이스 스캔을 줄임으로써 I/O 비용을 최소화할 뿐 아니라, 효과적인 탐사 방법을 사용하여 계산 비용 또한 줄여준다.

그림 3.1은 SPADE 알고리즘을 보여주고 있다. F_1과 F_2를 구한 후, F_2를 전위(pre*f*ix) 기반의 독립적인 동치 클래스로 나누고, 넓이 우선 탐사나 깊이 우선 탐사를 통해 각 클래스 내의 모든 빈발 시퀀스를 찾는다.

```
SPADE(min_sup, D):
    F1 = { frequent items or 1-sequences };
    F2 = { frequent 2-sequences };
    ε = { equivalence classes[X]θ1 };
    for all [X] ∈ ε do Enumerate-Frequent-Seq([X]);
```

[그림 3.1] SPADE 알고리즘

SPADE_V(SPADE Variation)는 위와 같은 SPADE 알고리즘의 몇 가지 특성을 이용하여 순회 패턴 탐사에 맞게 변형시켜 적용한 알고리즘이다.

우선, SPADE_V 알고리즘은 SPADE 알고리즘에서 제안한 수직 식별자 리스트 데이터베이스 개념을 적용한다. 이후부터는 간단하게 수직 데이터베이스라 하겠다. 그 구조는 각 시퀀스가 트랜잭션 데이터베이스 내에서

어느 위치에서 나타나는가에 대한 정보를 가지고 있고, 그러한 정보는 식별자 리스트 형태로 보유하고 있다. 이러한 수직 데이터베이스 형식을 사용하기 위해서는 전처리 과정이 필요하다. 그림 3.2는 그림 2.2의 수평 데이터베이스 형식을 수직 데이터베이스 형식으로 변환한 예제이다. 식별자 리스트에서 마지막 요소인 EID는 한 입력 트랜잭션에서 그 시퀀스가 발생한 첫 번째 항목에 대한 항목 리스트에서의 상대적인 위치이다. 예를 들어, 그림 3.2에서 항목 1은 트랜잭션 데이터베이스 내에서 2번 발생했다. 즉, CID가 2이고 TID가 2인 트랜잭션에서 그 입력 시퀀스의 첫 번째 위치에서 발생했다는 것을 나타내고, 그 다음은 CID가 1이고 TID가 1인 트랜잭션의 첫 번째 위치에서 그 항목이 발생했다는 것을 나타낸다. 이 자료구조를 이용하는 궁극적인 목적은 각 단계에서 지지도를 계산할 때 두 k-1 길이의 부분 시퀀스의 식별자 리스트를 단순히 조인하는 것으로 k-시퀀스의 지지도를 쉽게 얻을 수 있기 때문이다. SPADE_V 알고리즘의 각 단계별 자세한 사항은 다음과 같다.

Item	#. *of* Id-list	Id-list(CID, TID, EID)
1	2	(2 2 1) (1 1 1)
2	2	(2 2 2) (1 1 2)
3	6	(4 3 1) (4 2 1) (2 2 3) (1 3 1) (1 2 1) (1 1 3)
4	9	(4 3 9) (4 3 2) (4 2 3) (3 1 1) (2 2 4) (1 4 1) (1 3 2) (1 2 2) (1 1 4)
5	6	(4 3 3) (4 2 3) (3 1 2) (1 4 2) (1 3 3) (1 1 5)
6	6	(4 3 4) (4 2 4) (3 1 3) (1 4 3) (1 3 4) (1 1 6)
7	6	(4 3 5) (4 2 5) (4 1 5) (3 1 4) (2 1 1) (1 3 5)
11	1	(2 1 6)
12	3	(4 1 1) (2 2 6) (2 1 5)
13	5	(4 3 8) (4 1 2) (2 2 5) (2 1 4) (1 2 3)
14	4	(4 3 7) (4 1 2) (2 1 3) (1 2 4)
15	6	(4 3 6) (4 1 4) (3 1 5) (2 1 2) (1 3 6) (1 2 5)

[그림 3.2] 수직 데이터베이스 예제

1) 빈발 1-시퀀스(F_1)

SPADE_V 알고리즘은 전처리 과정에서 변환된 수직 데이터베이스를 입력으로 주어지면, 수직 데이터베이스를 한번 스캔하는 것으로 F_1을 구한다. 데이터베이스에 있는 각 항목에 대해 그 식별자 리스트를 디스크로부터 메인 메모리로 읽어 들여, 새로운 CID나 TID를 만날 때마다 그 항목의 지지도를 하나씩 증가시킨다.

2) 빈발 2-시퀀스(F_2)

SPADE 알고리즘에서는 F_2를 구하기 위해 두 가지 다른 방법을 제안했다. 하나는 전처리 과정을 통하여 F_2를 미리 구하는 방법이고, 다른 하나는 수직 데이터베이스 형식을 수평 데이터베이스 형식으로 변환하여 F_2를 계산하는 방법이다. 본 논문에서는 다른 순회 패턴 탐사 알고리즘과 형평성을 맞추기 위해 두 번째 방식을 사용한다. 수직 데이터베이스 형식을 수평 데이터베이스 형식으로 변화하기 위해서는 수직 데이터베이스 형식의 각 항목에 대한 식별자 리스트를 메인 메모리로 읽어 들이면서 변환해야 한다. 그런데 한 번의 데이터베이스 스캔으로 이렇게 변환하기 위해서는 결국 모든 수평 데이터베이스가 메인 메모리에 존재해야 한다. 그래서 본 논문에서는 수직 데이터베이스 형식의 파일을 스캔하여 변환하는 것이 아니라, F_1을 구할 때 읽혀진 식별자 리스트를 바로 F_2를 구하는데 사용한다. 즉 F_1 단계에서 읽혀진 각 항목에 대한 식별자 리스트를 F_2 단계에서 수평 데이터베이스 형태로 변환하여 F_2를 구한다. 이렇게 구할 수 있는 이유는 수직 데이터베

이스 형식의 식별자 리스트가 각 항목의 위치 정보도 포함하고 있기 때문이다.

3) 빈발 k-시퀀스(F_k)

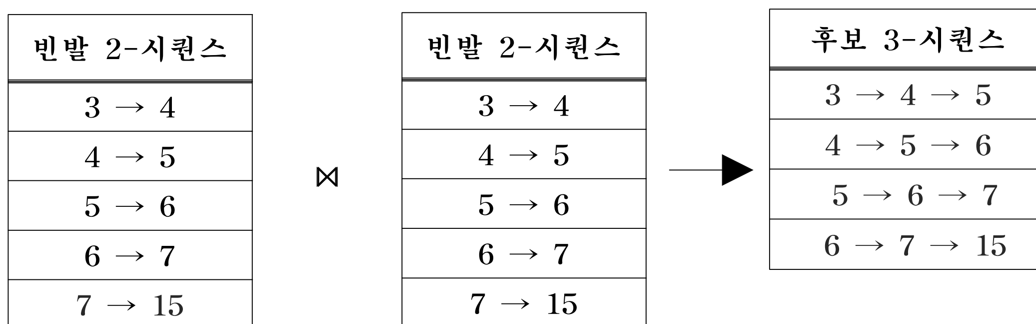
이전 단계에서 발견한 빈발 (k-1)-시퀀스를 이용하여 후보 k-시퀀스를 구하고, 후보 k-시퀀스에서 빈발 k-시퀀스를 구한다.

① 후보 k-시퀀스(C_k) 생성

SPADE 알고리즘에서는 F_{k-1}에서 두개의 시퀀스가 k-2 길이의 전위(pre*f*ix)가 같은 경우 동치 클래스에 속한다고 하며, 같은 동치 클래스에 속한 두 시퀀스를 조인해서 후보를 생성한다. 그러나 SPADE_V 알고리즘에서는 TPADE 알고리즘에서와 같은 방법으로 두 빈발 (k-1)-시퀀스를 조인한다. 즉 주어진 F_{k-1}에서 두 개의 시퀀스 중 한 시퀀스의 첫 번째 항목과 나머지 다른 시퀀스의 마지막 항목을 떨어뜨린 후, 남아있는 k-2 길이의 중위(in*f*ix)가 같은 경우 두 빈발 (k-1)-시퀀스를 조인시켜서 후보를 생성한다. 이러한 조인 방법은 본 논문에서 다루는 모든 순회 패턴 탐사 알고리즘에 적용된다.

지금까지는 두 빈발 (k-1)-시퀀스의 조인 조건에 대해 살펴보았다. 그렇다면 F_{k-1}의 모든 시퀀스를 어떻게 하면 효율적으로 조인시킬 수 있을까에 대한 문제에 직면하게 된다. 이에 SPADE_V에서는 정렬-합병 조인을 이용한다. 즉 F_{k-1}의 모든 빈발 k-시퀀스를 가리키는 포인터 배열을 만들고, 이것을 각 시퀀스의 첫 번째 항목으로 정렬시킨 후, F_{k-1}의 모든 빈발 k-시퀀스를 가리키는 포인터 배열의 복사본을 만들어 그 복사본을 다시 두

번째 항목으로 정렬 한다. 이렇게 만들어진 두 테이블에 정렬-합병 조인 알고리즘을 적용시킨다. 이는 k = 3일 때부터 반복 적용되는데, 매번 반복할 때마다 두 번씩 정렬하는 것이 아니라 k = 3일 때 처음에만 두 번 정렬하고, 그 다음 반복부터는 한번만 정렬한다. 왜냐하면 정렬-합병 조인 알고리즘의 특성상 정렬된 결과가 나오기 때문이다. 그림 3.3은 그 조인 예를 보이고 있다.



빈발 2-시퀀스
3 → 4
4 → 5
5 → 6
6 → 7
7 → 15

⋈

빈발 2-시퀀스
3 → 4
4 → 5
5 → 6
6 → 7
7 → 15

▶

후보 3-시퀀스
3 → 4 → 5
4 → 5 → 6
5 → 6 → 7
6 → 7 → 15

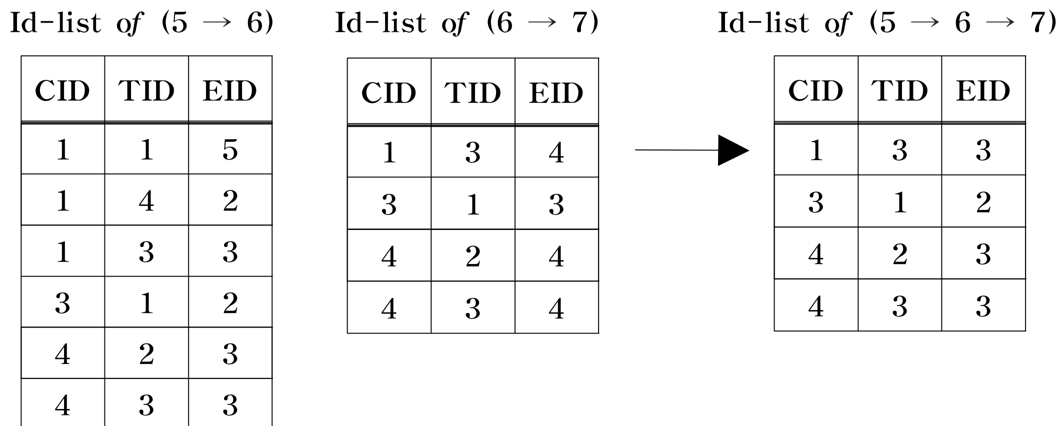
[그림 3.3] 빈발 2-시퀀스의 정렬 합병 조인

② 빈발 k-시퀀스(F_k) 발견

위 단계에서 생성된 후보 k-시퀀스는 두 빈발 (k-1)-시퀀스의 식별자 리스트를 접합시켜서 새로운 식별자 리스트를 얻고, 그 식별자 리스트 개수가 *min_sup*를 만족하면 F_k에 포함시킨다. 물론 이 부분에서도 한 트랜잭션 내의 중복 시퀀스 발생은 지지도 개수에 포함시키지 않아야 한다. 그리고 최대 빈발 시퀀스는 이 부분에서 아주 쉽게 찾을 수 있다. 후보 k-시퀀스가 *min_sup*를 만족하여 F_k에 포함되었을 경우, 그 후보 생성에 기여한 두 빈발 (k-1)-시퀀스는 최대 빈발 시퀀스가 될 수 없다. 왜냐하면 빈발 k-시퀀

스의 부분 시퀀스이기 때문이다. 그러므로 후보 k-시퀀스 생성에 기여하지 않고, 후보 k-시퀀스 생성에 참여했다 하더라도 실제 F_k에 속하지 않는 후보 k-시퀀스의 두 빈발 (k-1)-시퀀스는 모두 최대 빈발 시퀀스이다. 예를 들어 그림 2.3을 보면, 빈발 4-시퀀스인 (3 → 4 → 5 → 6)에 포함된 빈발 3-시퀀스 (3 → 4 → 5)와 (4 → 5 → 6)는 그림 2.4의 최대 빈발 시퀀스에서 제외된 것을 볼 수가 있다. 빈발 3-시퀀스 (3 → 4 → 5)와 (4 → 5 → 6)는 빈발 4-시퀀스 (3 → 4 → 5 → 6)의 부분 시퀀스이기 때문이다.

두 빈발 (k-1)-시퀀스의 식별자 리스트를 접합하는 과정은 다음과 같다. 두 시퀀스의 CID와 TID를 비교하여 같은 트랜잭션 내에 있을 경우 식별자 리스트에 저장되어 있는 트랜잭션 내의 위치 정보(EID)를 이용하여 연속적인지 비교하여 연속하면 새로운 후보 시퀀스의 식별자 리스트에 트랜잭션 내에서 앞쪽에 위치하는 시퀀스의 식별자 리스트를 추가한다. 그림 3.4는 그림 2.3에서의 빈발 2-시퀀스 (5 → 6)와 (6 → 7)를 접합하여 새로운 후보 3-시퀀스를 생성하는 예를 보이고 있다.



Id-list *of* (5 → 6)

CID	TID	EID
1	1	5
1	4	2
1	3	3
3	1	2
4	2	3
4	3	3

Id-list *of* (6 → 7)

CID	TID	EID
1	3	4
3	1	3
4	2	4
4	3	4

Id-list *of* (5 → 6 → 7)

CID	TID	EID
1	3	3
3	1	2
4	2	3
4	3	3

[그림 3.4] 식별자 리스트 접합 과정

2. GSP_V 알고리즘

GSP(Generalized Sequential Patterns)는 시퀀스의 길이(k)를 하나씩 증가시키면서 빈발 k-시퀀스를 구하는데 그때마다 반복적으로 데이터베이스를 스캔한다. GSP 알고리즘의 기본적인 구조는 그림 3.5와 같다. 그 첫 번째 단계는 데이터베이스를 한번 스캔하는 것으로 각 항목에 대한 지지도를 구하고 F_1을 결정한다. 그 다음부터는 이전 단계에서 구해진 빈발 $(k-1)$-시퀀스를 사용해서 후보 k-시퀀스를 구하고, 데이터베이스를 한번 스캔해서 각 후보 k-시퀀스의 지지도를 계산한 후 최소 지지도를 만족하는 후보 k-시퀀스만 F_k에 추가한다. 이러한 수행은 더 이상 빈발 시퀀스가 발견되지 않을 때까지 반복 수행한다.

```
F1 = { frequent 1-sequences };
for ( k = 2; Fk-1 ≠ ∅; k = k + 1 ) do
    Ck = Set of candidate k-sequences;
    for all input-sequences ℰ in the database do
        Increment count of all a ∈ Ck contained in ℰ;
    Fk = {a ∈ Ck | a.sup ≥ min_sup };
Set of all frequent sequences = ∪k Fk;
```

[그림 3.5] GSP 알고리즘

GSP_V(Generalized Sequential Patterns_Variation) 알고리즘도 GSP와 같은 원리이다. 그러나 후보 시퀀스 생성부분에서 다르다. 순차 패턴 탐사의 경우 한 타임스탬프에 여러 개의 항목이 존재할 수 있지만, 순회 패턴 탐사의 경우 한 타임스탬프에 한 항목만이 존재할 수 있다. 그렇기 때문에 후보를 생성할 때 이전 단계에서 발견된 두 빈발 시퀀스를 조인하는 부분에 있어서 차이가 난다. 그리고 순차 패턴 탐사의 경우 조인 한 후에 이전 단계의 빈발 시퀀스 집합에 포함되지 않는 후보 시퀀스를 제거시키는 전지(Pruning) 단계가 필요한 반면, 순회 패턴 탐사는 전지 단계가 필요 없다. 그리고 한 입력 트랜잭션에서 부분 시퀀스가 후보 시퀀스인지 확인하는 수를 줄이기 위해 해시 트리 자료구조를 사용한다. 해시 트리에 후보 시퀀스의 추가는 루트부터 시작해서 각 항목에 대해 해시 함수를 적용한 후 해당 버킷에 연결된 다음 레벨의 노드를 따라가는 방법으로 각 항목에 대해 순차적으로 모두 적용해서 마지막 레벨의 노드에 도달할 때까지 반복적으로 적용한다. 그래서 그 해시트리의 마지막 레벨의 노드에 후보를 추가한다. GSP_V 알고리즘의 각 단계별 자세한 사항은 다음과 같다.

1) 빈발 1-시퀀스(F_1)

트랜잭션 데이터베이스를 한번 스캔하는 것으로 해시 트리의 루트를 생성한다. 즉, 각 항목에 대해 해시 함수를 적용해서 해당 버킷을 구한 후, 대응되는 버킷에 해당 항목이 이미 존재하면 같은 트랜잭션의 중복된 항목인지 확인해서 중복이 아닌 경우 그 지지도를 증가시키고, 대응되는 버킷에 해당 항목이 없는 경우 그 항목을 새로 만든다. 이러한 방법으로 각 항목에 대해 지지도를 구한 후, 최소 지지도를 만족하지 못한 항목만 해시 트리에

서 제거한다. 결국, 빈발 1-시퀀스만 남게 된다.

2) 빈발 2-시퀀스(F_2)

GSP 알고리즘은 k = 2일 때부터 시작해서 반복적으로 수행한다. 즉 F_{k-1}을 self-join해서 후보 k-시퀀스를 생성한 후 실제 F_{k-1}에 없는 후보 k-시퀀스는 전지 단계에서 제거한다. 이와 달리 GSP_V 알고리즘에서는 k = 3일 때부터 반복하고, k = 2인 단계에서는 C_2를 미리 생성하지 않는다. F_1을 self-join하는 것으로 C_2를 생성한다면 실제 트랜잭션 데이터베이스에 존재하지 않는 매우 많은 수의 후보 2-시퀀스가 생성될 수 있다. 그러므로 GSP_V에서는 트랜잭션 데이터베이스를 스캔하면서, 연속적인 두 항목이 모두 F_1에 존재하는 경우만 후보 2-시퀀스로 생성하고 그에 대한 지지도를 계산한다. 미리 존재하지 않는 후보 시퀀스까지 해시 트리에서 만들었다가 다시 제거할 필요가 없기 때문이다. 이는 실제 트랜잭션 데이터베이스에 존재하는 시퀀스만 생성하므로 더 효율적이다. 트랜잭션 데이터베이스의 스캔이 끝나면 해시 트리의 두 번째 레벨에 있는 모든 노드를 스캔해서 최소 지지도를 만족하지 못한 후보 2-시퀀스만 제거한다. 결국, 해시 트리의 두 번째 레벨의 노드에는 F_2만 남게 된다.

3) 빈발 k-시퀀스(F_k)

F_{k-1}에서 두 개의 시퀀스 중 한 시퀀스의 첫 번째 항목과 나머지 다른 시퀀스의 마지막 항목을 떨어뜨린 후, 남아있는 k-2 길이의 중위(in*fix*)가 같은

경우 두 빈발 (k-1)-시퀀스를 조인시켜서 후보 k-시퀀스를 생성한다. 트랜잭션 데이터베이스를 스캔해서 각 후보 k-시퀀스의 지지도를 계산한다. 트랜잭션 데이터베이스의 스캔이 끝나면 각 k번째 레벨의 노드를 스캔해서 최소 지지도를 만족하지 못한 후보 k-시퀀스만 제거한다. 결국, 해시트리의 마지막 레벨 노드는 F_k만 남는다.

3. FS_V 알고리즘

FS 알고리즘에 대해서는 위의 관련 연구 분야에서 이미 언급했다. 위에서 언급된 것은 알고리즘의 특징과 전반적인 수행 과정이고, 이 부분에서는 FS_V(Full Scan_Variation) 알고리즘에서 어떻게 데이터베이스의 크기를 점증적으로 줄였는지에 대해 자세히 설명하고, FS_V 알고리즘의 전반적인 수행 과정에 대해 설명한다.

FS_V 알고리즘은 후보 시퀀스의 지지도 계산을 위해 트랜잭션 단위로 트랜잭션 데이터베이스를 읽어 들인다. 트랜잭션의 각 후보 k-시퀀스에 대해 지지도를 계산한 후, 다음 단계에 사용될 시퀀스만 데이터베이스에 기록한다. 다음 단계에 사용될 시퀀스의 조건은 후보 k-시퀀스를 포함해야 하고, 그 시퀀스의 길이가 k+1이어야 한다. 이와 같은 조건은 충분조건이 아닌 필요조건이다. 조건을 만족한 시퀀스를 데이터베이스에 기록하는 방법은 두 가지가 있다. 하나는 비트 연산을 사용하는 것이고, 다른 하나는 덧셈 연산을 사용하는 것이다.

예를 들어 보자. 빈발 2-시퀀스 발견 단계에서 후보 2-시퀀스의 지지도

를 계산할 때, 한 입력 트랜잭션으로 ABCDE가 있고, $a[i]$는 한 트랜잭션에서 i번째 항목의 발생 여부를 나타낸다고 하자. 이때 비트 연산의 경우, 우선 $a[i]$를 모두 0으로 초기화 하고, 트랜잭션을 조사해서 각 후보 2-시퀀스에 대해 비트 OR 연산을 한다. 만약 AB, BC가 후보 2-시퀀스이고, CD, DE는 후보 2-시퀀스가 아닐 경우, $a[i]$는 1, 1, 1, 0, 0으로 설정되고, 1로 설정된 ABC는 후보 시퀀스에도 포함되고, 그 시퀀스의 길이가 2보다 크기 때문에 데이터베이스에 기록된다. 그 트랜잭션의 나머지 DE는 필터링 된다. 또 다른 예로써, 이번엔 AB, DE가 후보 2-시퀀스이고, BC, CD가 후보 2-시퀀스가 아닐 경우, $a[i]$는 1, 1, 0, 1, 1로 설정되고, 위의 조건을 만족하는 시퀀스가 없으므로 이 트랜잭션은 데이터베이스에 기록되지 않는다. 그러므로 전체 트랜잭션 데이터베이스에서 트랜잭션의 수가 하나 감소하여 데이터베이스의 사이즈가 축소된다.

이번에는 덧셈 연산을 고려해 보자. 덧셈 연산의 경우에는 $a[i]$에 각 항목의 발생 빈도를 계산한다. 각 후보 시퀀스에 대해 덧셈 연산을 한 후 그 설정 값이 처음에 1, 2 순서로 시작해서 다음 1을 만날 때까지의 길이가 k보다 클 경우 그 시퀀스를 데이터베이스에 기록한다. 만약 AB, BC, DE가 후보 2-시퀀스이고, CD가 후보 2-시퀀스가 아닐 경우, $a[i]$는 1, 2, 1, 1, 1로 설정되고, 1, 2, 1에 대한 ABC만 데이터베이스에 기록한다. 나머지 DE는 필터링 된다. 또 다른 예로써, AB, CD가 후보 2-시퀀스이고, BC, DE가 후보 2-시퀀스가 아닐 경우, $a[i]$는 1, 1, 1, 1, 0으로 설정되고, 조건을 만족하는 시퀀스가 하나도 없으므로 이 트랜잭션은 데이터베이스에 기록되지 않는다. 트랜잭션 데이터베이스에서 한 트랜잭션이 제거된다. 이와 같이 비트 연산이나 덧셈 연산과 같은 방법으로 다음 단계에서 사용될 시퀀스만 데이터베이스에 기록할 경우, 각 단계마다 조건에 만족하지 못한 시퀀스

는 제거되고, 그에 따른 데이터베이스의 사이즈는 점차 줄어들게 된다. 즉, FS_V는 트랜잭션 데이터베이스의 크기를 줄이는데 있어서, 각 트랜잭션의 사이즈를 줄일 뿐만 아니라, 그 데이터베이스에서 트랜잭션의 개수도 줄인다. 각 단계별로 트랜잭션 데이터베이스의 사이즈가 점점 줄어들면, 후보 시퀀스의 지지도를 계산하기 위해 트랜잭션 데이터베이스를 스캔할 때 그만큼 디스크 I/O가 줄어들게 되므로 전체 트랜잭션 데이터베이스를 반복적으로 스캔하는 것보다 더 좋은 성능을 보이게 된다.

FS_V알고리즘은 FS 알고리즘과 달리 다음 단계의 후보 생성을 위한 해시 테이블을 생성하지 않는다. 그리고 해시 트리를 사용하지 않고 각 단계별로 해싱 기법은 사용한다. 해싱 값은 각 단계별 k개의 항목을 모두 더해서 해시 테이블의 사이즈로 나눈 나머지로 구해진다. 다음은 FS_V 알고리즘을 어떻게 구현했는지 각 단계별로 자세히 알아본다.

1) 빈발 1-시퀀스(F_1)

우선, 다른 순회 패턴 탐사와 마찬가지로 트랜잭션 데이터베이스를 한번 스캔하는 것으로 F_1을 구한다. 즉 각 항목에 대해 해시 함수를 적용해서 대응되는 버킷에 해당 항목이 존재하면 같은 트랜잭션에서의 항목인지 중복을 확인 한 후, 중복 발생이 아닌 경우 그 지지도를 증가시킨다. 만약 대응되는 버킷에 해당 항목이 존재하지 않을 경우 해시 테이블에 항목을 추가한다.

2) 빈발 2-시퀀스(F_2)

기존의 FS 알고리즘은 C_2를 생성할 때 이전 단계에서 생성된 해시 테이블(H_2)을 참조하여 C_2를 생성한 후 트랜잭션 데이터베이스를 스캔하여 지지도를 구한다. 그러나 FS_V는 이전 단계에서 미리 해시 테이블을 생성하지 않고, F_1의 각 항목을 인덱스로 사용하여 이차원 배열로 후보 2-시퀀스를 나열한 후 트랜잭션 데이터베이스를 스캔하여 지지도를 계산한다. 트랜잭션 데이터베이스를 스캔할 때 트랜잭션 단위로 스캔하며, 트랜잭션내의 각 후보 2-시퀀스에 대해 지지도 계산이 끝나면 다음 단계에 사용될 시퀀스만 데이터베이스에 기록한다. 트랜잭션 데이터베이스 스캔이 끝나면 각 후보 2-시퀀스에 대해 최소 지지도를 만족하는 후보 2-시퀀스만 F_2에 포함시킨다.

3) 빈발 k-시퀀스(F_k)

F_2가 구해지면 그 다음부터는 더 이상 후보 시퀀스나 빈발 시퀀스가 발견되지 않을 때까지 반복 수행한다. 후보 k-시퀀스의 생성은 주어진 F_{k-1}에서 두 개의 시퀀스 중 한 시퀀스의 첫 번째 항목과 나머지 다른 시퀀스의 마지막 항목을 떨어뜨린 후, 남아있는 k-2 길이의 중위(in*fix*)가 같은 경우 두 빈발 (k-1)-시퀀스를 조인시켜서 후보 k-시퀀스를 생성한다. 후보 k-시퀀스의 지지도 계산은 이전 단계어서 축소된 데이터베이스를 트랜잭션 단위로 읽어 들여, 각 k 길이의 시퀀스가 후보 k-시퀀스인지 확인해서 후보 시퀀스인 경우 중복 발생인지 확인한다. 중복 발생이 아닌 경우 그 지지도를 증가시키고, 다음 단계에 사용될 시퀀스인지를 계산하기 위해 비트 연산이나 아니면

덧셈 연산을 수행한다. 그래서 한 트랜잭션의 지지도 계산이 끝나면 비트 연산이나 덧셈 연산 수행을 참조하여 다음 단계에 사용될 (k + 1)-시퀀스만 데이터베이스에 기록한다. 트랜잭션 데이터베이스 스캔이 끝나면 각 후보 k-시퀀스에 대해 최소 지지도를 만족하는 후보 k-시퀀스만 F_k에 포함시킨다.

4. AVL 알고리즘

대량의 데이터베이스에서 모든 빈발 시퀀스를 발견하는 것은 그 탐사 공간이 아주 크기 때문에 꽤 힘든 작업이다. 모든 가능한 부분 시퀀스의 폭발적인 수가 조사될 수 있기 때문이다. 예를 들어, 발견된 F_1의 개수가 100개일 경우 Apriori-기반의 알고리즘들은 100 * 100개의 후보 시퀀스를 생성할 수 있고, 각 후보 시퀀스의 길이가 하나씩 증가할 때마다 데이터베이스를 한번 스캔한다. 만약 시퀀스의 길이가 6일 경우 6번의 데이터베이스를 스캔해야 한다. 또한 후보 시퀀스가 많이 생성될수록 여러 자원 사용면에서 비효율적이다. 이에 데이터베이스의 스캔 횟수를 줄이면서 후보시퀀스를 적게 생성할 수 있는 방법에 대해 연구하였다. AVL(Array and Vertical List) 알고리즘은 배열 자료구조와 수직 데이터베이스 개념을 이용하여 복합 적용한 알고리즘이다.

AVL은 다른 순회 패턴 탐사 알고리즘과 마찬가지로 점증적인 방법으로 탐사한다. 입력으로 트랜잭션 데이터베이스 D와 최소 지지도(*min_sup*)가 주어졌을 때, AVL 알고리즘은 시퀀스의 길이(k)를 하나씩 증가시켜가면서 빈발 시퀀스를 발견하고, 모든 빈발 시퀀스들 중 최대 빈발 시퀀스만을 결과로 출력한다. 그 알고리즘은 그림 3.6과 같다.

```
AVL(min_sup, D)
    F1 = { frequent 1-sequence }
    F2 = { frequent 2-sequence }
    k = 3;
    do {
            Decompose( Fk-1 );
            Fk = Enumerate( );
            k++;
    } while ( Fk-1 ≠ ∅ )
```

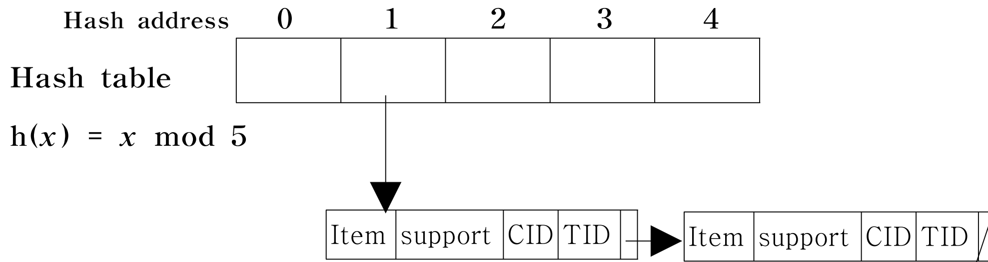
[그림 3.6] AVL 알고리즘

AVL은 크게 F_1 발견 단계, F_2 발견 단계 그리고 F_3부터는 반복단계로 구성된다. 첫 번째 단계는 입력 트랜잭션 데이터베이스에서 데이터를 읽어 들여 on-the-*f*ly방식으로 해시테이블을 사용하여 결정한다. 두 번째 단계로 C_2의 생성은 F_1을 sel*f*-join하는 것으로 생성되고, 생성된 C_2를 이차원 배열로 방식으로 전개한다. 그래서 F_1을 입력 필터로 사용하여 입력 트랜잭션에서 F_1에 속한 항목들만 C_2의 지지도 계산에 참여하도록 한다. 이러한 방식으로 F_2를 발견한다. 그 마지막 단계는 k = 3일 때부터 시작하여 더 이상 빈발 시퀀스가 생성되지 않을 때까지 반복 수행한다. F_3부터는 SPADE 알고리즘에서 제안한 수직 데이터베이스 형식 구조를 사용한다. 그러기 위해서는 C_2를 생성할 때 C_3을 생성하기 위한 형태로 저장해야 한다.

TPADE에서 제안했던 조인 방법을 사용하여 C_k를 구하고, 그에 대한 지지도 계산은 단순히 접합된 식별자 리스트의 개수를 확인하는 것으로 계산된다. AVL 알고리즘의 각 단계별 자세한 내용은 다음과 같고, F_3 부터는 SPADE_V 알고리즘과 같기 때문에 해당 부분을 참조한다.

1) 빈발 1-시퀀스(F_1)

입력으로 트랜잭션 데이터베이스와 최소 지지도가 주어지면, F_1은 트랜잭션 데이터베이스를 한번 스캔하는 것으로 얻어진다. 그림 2.2의 트랜잭션 데이터베이스가 입력으로 주어졌을 때, 그림 2.3은 빈발 1-시퀀스가 탐사됨을 보이고 있다. 즉 트랜잭션 데이터베이스에서 항목을 하나씩 읽어 들여 단순히 그 항목의 개수를 증가시키면 된다. 물론 한 트랜잭션의 항목 리스트는 같은 항목이 중복되어 발생할 수 있다. 이 경우에 한 트랜잭션의 중복된 항목에 대한 지지도 계산은 같은 트랜잭션에서 한 항목에 대해 여러 번 중복이 발생하더라도 한번만 계산된다. 이러한 작업을 할 때 각 항목을 읽어 들여 메모리에 어떻게 저장할 것인가에 대해 고려해야 한다. 단순히 각 항목 값 자체를 인덱스로 사용하여 일차원 배열에 저장 할 경우 메모리 낭비가 심하게 발생할 수 있다. 예를 들어 입력되는 항목이 연속적인 값이 아닐 경우, 즉 한 항목의 값이 다른 항목에 비해 아주 크거나 아주 작을 경우 그 항목을 수용할 수 있을 만큼 배열 사이즈를 결정해야 한다. 이는 메모리 낭비이다. 그리고 배열에서 그 항목을 찾기 위한 비용도 예상해야 한다. AVL 알고리즘에서는 이러한 문제를 해결하고자 해시 테이블을 사용한다. 즉 항목에 해시 함수를 적용시켜 그 해당 해시 테이블의 버킷에 연결시키면 된다. 이렇게 함으로 좀 더 빠르게 처리될 수 있다.



[그림 3.7] F_1에서의 해시 테이블

그림 3.7은 해시 테이블 자료구조를 보이고 있다. 해시 테이블의 각 버킷은 하나의 노드를 가리키는 포인터이고, 한 노드는 항목, 지지도, CID, TID 그리고 그 버킷에 충돌이 발생할 경우 체인을 만들어 연결할 수 있는 포인터로 구성된다. 여기에서 CID와 TID는 한 트랜잭션 내에 해당 항목이 중복이 발생했는지를 확인하기 위한 것이다. 트랜잭션 데이터베이스에서 한 항목씩 읽어 들여 그 항목에 해시 함수를 적용시키고, 해당 버킷에 연결된 노드에 해당 항목이 있는지 확인한다. 해당 항목이 있을 경우 그 항목의 지지도를 계산하고 항목이 없을 경우 새로운 노드를 만들어 해당 버킷에 체인으로 연결한다. 지지도 계산은 새로운 CID이거나 TID를 만날 경우 그 지지도를 증가시킨다. 그리고 CID와 TID를 기록한다. 이런 방식으로 트랜잭션 데이터베이스의 모든 항목들을 적용하고, 트랜잭션 데이터베이스의 스캔이 끝나면 각 항목의 지지도가 *min_sup*보다 크거나 같은지 확인하여 이를 만족하는 항목만 F_1에 포함시킨다. 그리고 해시 테이블에 연결된 노드 생성은 새로운 항목이 들어올 때마다 하나씩 생성하는 것이 아니라 데이터 풀 개념을 사용하여 한꺼번에 임의적인 개수로 할당 받는다. 필요할 때마다 데

이터 풀에서 획득한다. 할당 받은 개수가 부족할 경우 추가적으로 일정한 개수만큼 제한적인 회수로 더 할당을 받는다. 이렇게 함으로 성능을 조금이나마 향상시킬 수 있다.

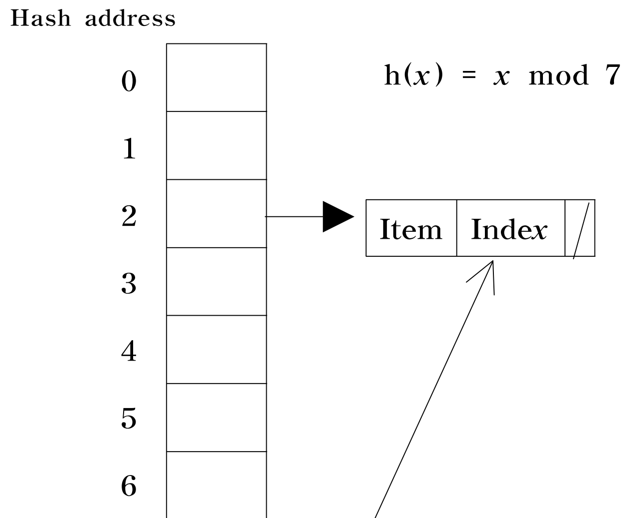
2) 빈발 2-시퀀스(F_2)

① 후보 2-시퀀스(C_2) 생성

C_2의 생성은 F_1을 self-join하는 것으로 생성할 수 있다. 위에서도 언급했듯이 C_2 생성 개수는 폭발적으로 증가할 수 있다. 그러므로 C_2에서 비용을 줄이는 것은 순회 패턴 탐사의 전반적인 성능을 크게 향상시키는 결과를 낳는다. 그 만큼 F_2 단계에서 병목 현상이 크다는 것을 알 수 있다. 그렇다면 이 단계에서는 어떤 자료구조를 사용해야 효율적으로 F_2를 탐사할 수 있을까? 해시 트리 자료 구조는 일반적으로 빠른 지지도 계산을 위해 사용 된다[17]. 기존 연구를 살펴보면 연관 규칙 탐사 알고리즘 중에 하나인 Apriori 알고리즘[12]와 순차 패턴 탐사 알고리즘 중에 하나인 GSP[2], AprioriAll[7], AprioriSome[7], DynaminSome[7] 알고리즘들도 해시 트리를 사용한다. 기존의 알고리즘에서 해시 트리 자료구조를 많이 사용하지만, 비용이 많이 든다. 즉, 트랜잭션 데이터베이스를 스캔하여 C_2에서 만들어진 해시 트리로 L_2를 결정하기 위한 단계는 많이 비용이 든다[8]. C_2까지 만들어진 해시 트리는 그 최악의 경우 루트 노드의 해시 테이블 사이즈 개수만큼 새로운 해시 테이블을 만들어야 하기 때문에 저장 공간이 많이 낭비된다. 해시 트리의 높이가 높아질수록 낭비되는 공간은 더욱 증가한다.

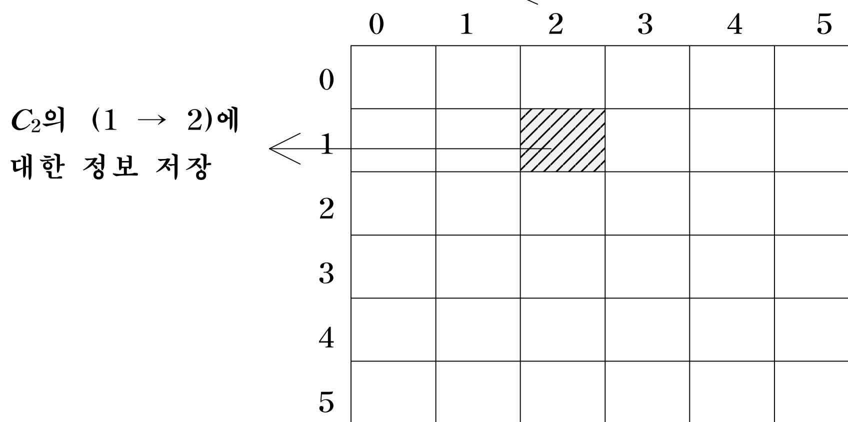
그러므로 AVL 알고리즘에서는 F_1을 조인하는 것에 의해 생성된 C_2를

이차원 배열 방식으로 전개한다. $|F_1|$ * $|F_1|$의 개수만큼 이차원 배열에 저장하기 위해 해싱 기법을 적용한다. 그림 3.8은 해싱 기법을 적용하기 위한 Lookup 해시 테이블을 나타내고, 그림 3.9는 F_1의 자기 자신을 조인한 형태의 C_2를 나타낸다. 그림 3.8의 Lookup 해시 테이블의 버킷은 항목과 인덱스 정보를 갖는 한 노드를 가리키는 포인트로 이루어져 있다. F_1을 한번 스캔해서 각 항목에 해싱을 적용하여 해당 버킷의 노드에 항목을 저장하고, 그 인덱스를 설정한다. 이렇게 만들어진 Lookup 해시 테이블은 F_1의 각 항목에 대해 이차원 배열을 참조할 수 있는 인덱스로 사용된다. 이는 C_2에서 F_2를 결정할 때 지지도를 빠르게 계산하기 위해서 사용되고, 또한 실제 $|F_1|$ * $|F_1|$의 개수만큼 배열을 사용하기 위해서이다. 위의 F_1 탐사에서도 언급했듯이 항목 값 자체를 인덱스로 사용하는 이차원 배열을 만든다면 메모리 낭비가 많을 것이기 때문이다. 이와 같이 이차원 배열을 사용함으로 빠른 지지도 계산을 할 수 있다.



[그림 3.8] C_2 생성에서의 Lookup 해시 테이블

Lookup 해시 테이블의 한 노드에서 해당 항목에 대한 인덱스 정보



[그림 3.9] C_2 생성에서의 이차원 배열

② 빈발 2-시퀀스(F_2) 발견

트랜잭션 데이터베이스를 한번 스캔하는 것에 의해 F_2를 구할 수 있다. 이 단계는 F_1을 해시 필터로 사용하여 입력 트랜잭션에서 F_1에 속한 항목들만이 C_2의 지지도 계산에 참여하도록 하였다. 즉, 입력 트랜잭션을 읽어들여 각 항목이 Lookup 해시 테이블에 존재하는지 확인하고, 존재하면 그 인덱스를 얻는다. 이 부분에서 연속된 두 항목이 각각 Lookup 해시 테이블에 존재해야 만이 지지도 계산에 참여된다. 이는 순차 패턴 탐사와 달리 순회 패턴 탐사는 시퀀스를 이루는 항목 리스트가 한 번에 하나의 항목만을 갖는 항목들의 리스트로 이루어져 있고, 그 시퀀스의 부분 시퀀스는 연속해서 발생해야하는 특징을 갖기 때문이다. 예를 들어, 한 시퀀스 (3 → 5 → 8)가 있고, 이 시퀀스에서 항목 5는 F_1 집합에 포함되어 있지 않다고 하자. 이 입력 트랜잭션에 대한 C_2의 지지도를 계산할 때 시퀀스 (3 → 8)는 시퀀스 (3 → 5 → 8)의 후보 2-시퀀스가 될 수 없다. 왜냐하면 연속적인 발생이 아니기 때문이다. 즉, 부분 시퀀스 정의에 위배된다. 그러므로 이런 시퀀스는 C_2의 이차원 배열에서 계산될 수 없다. 이 단계에서도 마찬가지로 중복을 확인하는 작업이 필요하다. 왜냐하면 위에서 언급했듯이 중복은 지지도 계산에 포함되지 않아야하기 때문이다.

입력 트랜잭션에 존재하는 모든 C_2는 그 생성단계에서 만들어진 이차원 배열의 해당 인덱스가 가리키는 곳에 그 지지도를 증가키시고, 입력 트랜잭션들이 모두 검사되고 나면, 이차원 배열을 순회하면서 그 지지도가 min_sup를 만족하는 후보 2-시퀀스만 F_2 집합에 포함시킨다. 이 단계의 특징은 F_1을 해시 필터로 사용해서 F_1에 존재하는 항목만 C_2에 참여하도록 하고, C_2를 이차원 배열로 전개함으로써 F_2를 빠르게 발견할 수 있다는 것

이다. 메인 메모리의 사이즈가 그리 크지 않았을 때는 C_2의 이차원 배열을 메인 메모리에 저장한다는 것은 불가능했다. 그러나 최근에는 하드웨어의 급속한 발전으로 이를 수용할 만한 충분한 기반이 되었기 때문에 이러한 작업이 가능해졌다. 그만큼 빠른 계산을 유도해 낼 수 있다는 것을 의미한다.

AVL 알고리즘에서는 C_3 생성부터 SPADE 알고리즘에서 제안한 수직 식별자 리스트 데이터베이스 개념을 사용하기 때문에 C_2 생성 단계에서 C_3 형태에 맞는 구조로 자료로 저장시켜야 한다. 그 상세한 내용은 앞 단락의 SPADE_V에서 참조한다.

Ⅳ. 실험 결과 및 분석

1절에서는 실험 환경 및 실험 데이터의 특성에 대해 설명하고, 2절에서는 각 구현된 순회 패턴 알고리즘에 교통 카드 트랜잭션 데이터베이스를 적용한 실험 결과와 그 성능을 비교 분석해 본다.

1. 실험 환경 및 실험 데이터 특성

각 순회 패턴 탐사 알고리즘들의 성능을 측정하기 위해 3.0 GHz Intel P-4 CPU, 1 GB의 메모리, Windows XP Professional 운영체제 그리고 MS Visual C++ 6.0 언어를 사용하여 실험하였다.

실험 데이터는 수도권 서울 지역에서 2004년 10월 27일 하루 동안 교통 카드를 사용한 승객들의 사용 내역에 관한 트랜잭션 데이터베이스의 일부이다. 교통 카드에 의한 트랜잭션 데이터베이스의 각 트랜잭션은 한 승객의 승차역과 하차역 그리고 환승 회수에 관한 정보를 포함하고 있다. 그 승차역과 하차역 사이의 중간 이동 경로에 대한 정류장들은 전처리 프로세스에 의해 채워졌으며, 그리고 버스 정류장 번호와 지하철역 번호의 사용범위가 서로 상충되어 이를 구분하기 위해 지하철역 번호를 2000000 단위로 변경하였다. 이에 대한 자세한 사항은 [18]을 참조한다. 본 논문에서의 실험 데이터는 [18]에서 생성한 데이터를 이용한다.

본 논문에서 사용한 트랜잭션 데이터베이스는 순회 패턴 탐사를 수행하기 위해 사전 처리된 데이터이다. 한 트랜잭션은 고객 식별자(CID), 트랜잭

션 식별자(TID), 환승 회수, 이동 정류장(항목) 수, 정류장(항목) 리스트로 구성된다. 교통 카드를 사용한 승객의 트랜잭션은 승객이 승차하여 하차할 때까지 순차적으로 경유한 정류장(항목)들에 관한 정보를 포함하고 있다. 예를 들어 그림 4.1의 경우를 보자. 그림 4.1은 전처리 프로세스에 의해 생성된 트랜잭션 데이터베이스의 일부이다. 그 네 번째 줄의 경우, CID가 3967754이고 TID가 95인 승객은 환승을 하지 않았고, 다섯 정류장을 이동했으며 정류장 번호가 2000000 이상이므로 지하철을 이용했다는 것을 알 수 있다. 그 승객은 지하철역 번호가 2002522인 오목교역에서 승차해서 목동역, 신정역, 까치산역을 경유하여 지하철역 번호가 2002518인 화곡역에서 하차했다. 이러한 정보를 담고 있는 트랜잭션 데이터베이스는 그 전체 크기가 약 62MB이고, 479079개의 트랜잭션으로 구성되어 있다. 한 트랜잭션의 시퀀스 최대 길이는 150이고, 그 평균 길이는 14이다. 서로 구별되는 정류장(항목)의 총 수는 15115이다.

3967752 91 0 8 7331 7326 7347 7371 7400 7457
7476 12754

3967753 57 0 9 2001028 2001027 2001026 2001025
2001024 2001023 2000220 2000221 2000222

3967753 58 0 9 2000222 2000221 2000220 2001023
2001024 2001025 2001026 2001027 2001028

3967754 95 0 5 2002522 2002521 2002520 2002519
2002518

3967755 66 0 4 4271 4272 4273 4274

3967756 60 0 10 2000228 2000227 2000226 2000225
2000224 2000223 2000222 2000221 2000220 2000219

3967757 42 0 4 2000329 2000330 2000331 2000332

3967757 43 0 5 31571 31572 31573 31574 31575

3967757 44 0 5 4004 4005 4006 4007 4008

3967757 45 0 7 10078 8424 8425 10027 9986 9948
9895

[그림 4.1] 전 처리된 트랜잭션 데이터베이스

2. 실험 결과 및 성능 비교 분석

1) 실험 결과

다음은 본 논문에서 구현한 각 순회 패턴 탐사 알고리즘에 교통 카드 트랜잭션 데이터베이스를 적용한 실험 결과이다. 본 논문에서 구현한 모든 순회 패턴 탐사 알고리즘은 동일한 데이터와 동일한 변수가 주어질 경우 동일한 결과를 산출한다.

① 시퀀스 길이에 따른 탐사된 후보 시퀀스와 빈발 시퀀스

아래 표 4.1은 입력 데이터수가 100,000개이고, 최소 지지도 1.0%일 때 시퀀스 길이에 따른 탐사된 후보 시퀀스와 빈발 시퀀스의 개수를 나타낸다. 이 표에서 볼 수 있듯이 입력 트랜잭션 100,000개에는 서로 다른 항목이 14,641개가 존재하고, 그 트랜잭션 데이터베이스에서 각 항목에 대한 지지도를 계산하여 최소 지지도 1.0%를 만족하는 항목을 찾는다. 그 결과 14,641개의 항목 중에 291개의 항목만이 최소 지지도를 만족해서 F_1에 포함된다. 빈발 1-시퀀스 단계에서 상당히 많은 항목들이 최소 지지도를 만족하지 못하는 것을 알 수 있다. 그 만큼 후보 2-시퀀스를 생성할 때 그 처리 비용을 줄일 수 있다는 것을 의미한다. 그리고 후보 2-시퀀스는 F_1을 self-join해서 생성하는 것이 아니라, F_1을 이용하여 2차원 배열로 지지도를 계산할 저장 공간만 만들어 놓고, 실제 트랜잭션 데이터베이스에서 항목들을 읽어 들이면서 후보 2-시퀀스를 생성한다. 그래서 후보 2-시퀀스가

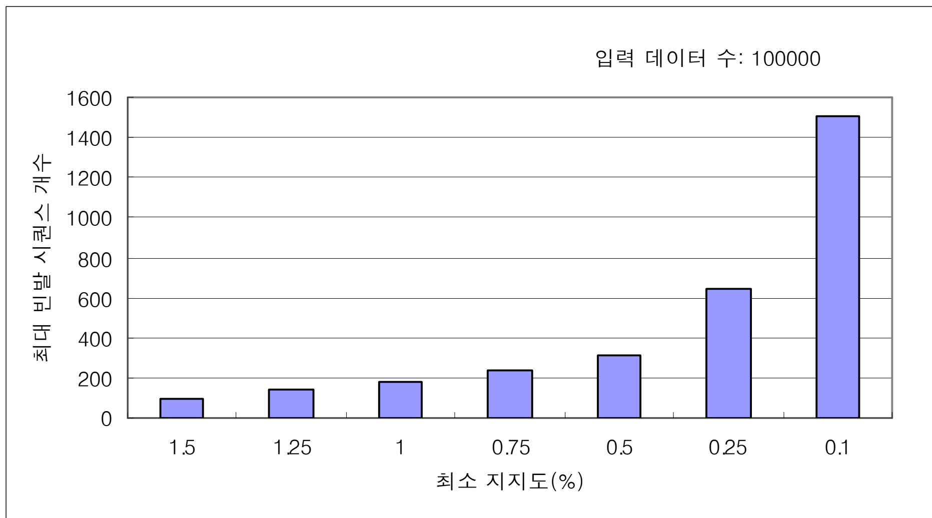
84,681개가 아닌 실제 638개인 것이다. 즉, F_1을 해시 필터로 사용하여 두 항목이 트랜잭션 데이터베이스에 연속적으로 존재할 때 후보 2-시퀀스가 되는 것이고, 이 후보 2-시퀀스들 중 최소 지지도를 만족하는 빈발 2-시퀀스의 수가 335개인 것이다. k = 3일 때부터 그 시퀀스 길이가 증가하면서 후보 시퀀스의 수와 빈발 시퀀스의 수가 줄어드는 것을 볼 수 있다. 아래 표 4.1의 모든 빈발 시퀀스 1,474개 중에서 탐사된 최대 빈발 시퀀스는 182개이다.

시퀀스 길이(k)	후보 시퀀스	빈발 시퀀스
1	14641	291
2	638	335
3	603	267
4	474	207
5	354	147
6	241	94
7	153	59
8	94	35
9	53	18
10	30	12
11	18	6
12	9	3

[표 4.1] 시퀀스 길이(k)에 따른 후보 시퀀스와 빈발 시퀀스의 개수
입력 데이터의 수: 100000, 최소 지지도: 1.0%

② 최소 지지도에 따른 탐사된 최대 빈발 시퀀스

그림 4.2는 최소 지지도에 따른 탐사된 최대 빈발 시퀀스의 수를 보이고 있다. 최소 지지도가 낮아질수록 더 많은 최대 빈발 시퀀스들이 탐사됨을 알 수 있는데, 이는 최소 지지도가 낮아질수록 더 많은 후보 시퀀스들이 최소 지지도 조건을 만족하고, 이로 인해 더 많은 빈발 시퀀스들이 발견되기 때문이다. 그러므로 그에 따른 더 많은 최대 빈발 시퀀스들이 탐사된다.



[그림 4.2] 최소 지지도에 따른 탐사된 최대 빈발 시퀀스

다음 표 4.2는 본 논문에서 구현한 순회 패턴 탐사 알고리즘에 교통 카드 트랜잭션 데이터베이스를 적용하여 얻은 최대 빈발 시퀀스의 일부이다. 입력 트랜잭션 수 100,000개와 최소 지지도 1.0%가 주어졌을 때, 탐사된 최대 빈발 시퀀스 중에 가장 긴 시퀀스를 나타내고 있다. 그 첫 번째 줄을

보면, 입력 트랜잭션 100,000개 중에 1,111개의 트랜잭션에서 해당 시퀀스를 발견했음을 알 수 있고, 이 발견된 시퀀스를 보면 지하철을 이용한 시퀀스임을 알 수 있다. 실제 이동 경로는 지하철역 번호가 2001806인 부평역에서 승차하여, 부개역, 송내역, 중동역, 부천역, 소사역, 역곡역, 온수역, 오류동역, 개봉역, 구일역을 경유하여 지하철역 번호가 2001701인 구로역에서 하차한 경로이다.

최대 빈발 12-시퀀스	지지도(개수)
2001806,2001815,2001805,2001822,2001804,2001814, 2001803,2001821,2001802,2001801,2001813,2001701	1,111
2001701,2001813,2001801,2001802,2001821,2001803, 2001814,2001804,2001822,2001805,2001815,2001806	1,007
2001815,2001805,2001822,2001804,2001814,2001803, 2001821,2001802,2001801,2001813,2001701,2001999	1,050

[표 4.2] 탐사된 최대 빈발 시퀀스 예제

2) 성능 비교 분석

다음은 본 논문에서 구현한 순회 패턴 탐사 알고리즘들에 대한 그 성능을 평가한 결과이다. 아래의 실험 결과는 각 알고리즘별로 10번 수행의 평균을 나타낸다.

① 입력 데이터 수에 따른 각 알고리즘의 실행 시간

그림 4.3은 각 알고리즘 별로 입력 데이터의 수에 따른 그 실행 시간을 보이고 있다. 그 그래프를 보면 GSP_V의 경우, 다른 알고리즘들에 비해 그 수행 시간이 가장 오래 걸렸음을 볼 수 있다. 이는 시퀀스의 길이가 증가할 때마다 전체 트랜잭션 데이터베이스를 반복적으로 스캔함으로써 그에 따른 디스크 I/O 비용도 많이 들고, 또한 전체 데이터베이스를 읽어 들이면서 그 데이터 하나하나를 처리해야 하는 비용도 많이 들기 때문에 그 수행 시간이 다른 알고리즘에 비해 높게 나온다.

FS_V도 GSP_V와 같이 시퀀스의 길이가 증가할 때마다 트랜잭션 데이터베이스를 반복적으로 스캔한다. 그러나 GSP_V와 달리 FS_V는 매 단계마다 데이터베이스를 필터링하는 기능이 있기 때문에 데이터베이스의 사이즈를 줄이고 다음 단계에서 사이즈가 줄어든 데이터베이스로 수행을 하기 때문에 그에 따라서 당연히 그 처리비용도 줄어들게 된다. 그러므로 GSP_V에 비해 FS_V의 성능이 우수함을 알 수 있다.

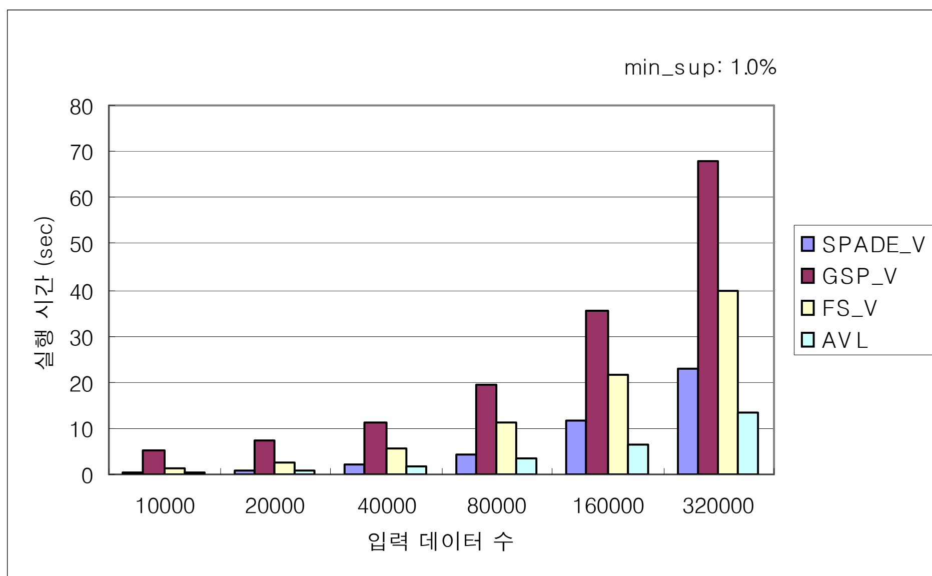
그리고 다음으로 GSP_V와 FS_V보다 SPADE_V의 수행 시간이 더 적게 걸림을 볼 수 있는데, 이는 SPADE_V의 경우 F_1에서 전체 트랜잭션 데이터베이스를 한번만 스캔하고 그 다음 단계부터는 메인 메모리 수행을 하기 때문에 트랜잭션 데이터베이스를 반복적으로 스캔하는 GSP_V와 FS_V보다 그 수행시간이 훨씬 적게 걸림을 알 수 있다. 그러나 SPADE_V의 경우 그림 4.4를 보면 알 수 있듯이 k = 1과 k = 2 단계에서 그 수행시간이 SPADE_V 알고리즘의 전체 수행시간을 좌우할 정도로 다른 단계에 비해 꽤 높음을 볼 수 있는데, 이는 SPADE_V 알고리즘이 데이터베이스의 형태를 변환시키는 과정이 있기 때문인 것으로 판단된다. 즉 SPADE_V는 F_1을

구할 때 수직 데이터베이스 형태의 데이터베이스를 읽어 들이는데, 이는 수평 데이터베이스를 즉, 원래의 트랜잭션 데이터베이스를 수직 데이터베이스로 변환시킨 것이고, 이 변환된 수직 데이터베이스는 사이즈는 원래의 트랜잭션 데이터베이스보다 증가된다. 왜냐하면 각 항목마다 CID와 TID에 대한 정보를 별도로 보유하고 있어야 하기 때문이다. 그러므로 F_1 구할 때 원래의 트랜잭션 데이터베이스 보다 더 큰 사이즈의 수직 데이터베이스를 읽게 되고, 그 만큼 그 처리비용도 증가한다. 그리고 F_1이 구해진 후에 F_2를 구하기 전에 다시 수직 데이터베이스를 수평 데이터베이스로 변환하는 작업을 수행한다. 그 변환 처리를 할 때 CID와 TID의 정렬 순서가 바뀌게 되고 후보 2-시퀀스의 중복발생을 확인할 때 그 수평 데이터베이스를 정렬시켜야 하는 오버헤드가 들기 때문에 그 만큼 처리 비용이 더 걸린다. SPADE_V는 F_1과 F_2 단계에서 너무 복잡한 자료구조를 사용하기 때문에 이 단계에서 병목 현상을 일으킨다. 그래도 SPADE_V는 GSP_V나 FS_V에 비해 그 수행시간이 적게 걸린다. 그러나 이 알고리즘은 한계가 있다. 메인 메모리 수행을 하기 때문에 데이터베이스가 클 경우 처리 할 수가 없다. 트랜잭션 데이터베이스가 작을 경우만 처리가 가능하다.

마지막으로 AVL 알고리즘의 경우, 다른 어떠한 알고리즘보다도 그 성능이 우수함을 보이고 있는데, 이는 그림 4.4에서도 볼 수 있듯이 k = 1일 때와 k = 2일 때에 FS_V와 SPADE_V보다 그 수행시간이 훨씬 적게 들고 k = 3일 때부터 FS_V와 GSP_V보다 수행 시간이 훨씬 적게 걸림을 볼 수 있다. 이는 F_1을 구할 때 해시 기법을 사용하고, F_2를 구할 때 C_2를 2차원 배열 방식으로 나열하여 데이터베이스를 스캔하면서 그 지지도를 2차원 배열에 계산하는 방법을 사용함으로 그 수행이 다른 알고리즘에 비해 우수하게 나온 것으로 보인다. 그리고 k = 3부터는 SPADE_V와 같이 수직 데이

터베이스 개념을 사용하여 메인 메모리 수행을 하기 때문에 그 시간이 아주 적게 걸린다. 따라서 전반적인 성능이 다른 알고리즘에 비해 월등히 우수함을 보인다. 순회 패턴 탐사의 경우 연관 규칙 탐사나 순차 패턴 탐사와 같이 F_1과 F_2를 구할 때 전체 수행의 병목현상을 일으키는데, AVL의 경우 F_1과 F_2를 효과적으로 탐사함으로 다른 알고리즘에 비해 우수하다. F_2까지만 구하면 그 이후부터는 처리해야할 데이터의 수가 많이 줄기 때문에 대부분의 경우 메인 메모리에서 수행이 가능하고, 그 만큼 효율적으로 빈발 시퀀스를 탐사할 수 있다.

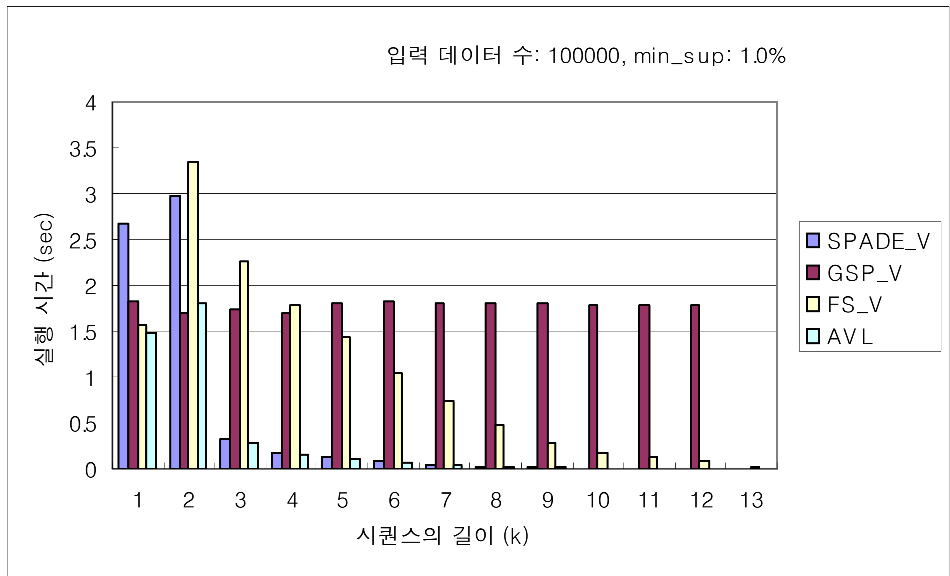
아래 그림 4.3의 그래프를 보면, 본 논문에서 구현한 모든 순회 패턴 탐사 알고리즘들은 입력 데이터 수가 증가할수록 그 증가 폭은 다르지만 그래도 그 수행시간이 선형적으로 증가함을 알 수 있다.



[그림 4.3] 입력 데이터 수에 따른 각 알고리즘의 실행 시간

② 시퀀스의 길이에 따른 각 알고리즘의 실행 시간

위에서도 언급했듯이 GSP_V와 FS_V는 시퀀스의 길이가 증가할 때마다 반복적으로 트랜잭션 데이터베이스를 스캔해야 한다. GSP_V의 경우 매 단계마다 반복적으로 전체 트랜잭션 데이터베이스를 스캔해서 처리하기 때문에 그 만큼 디스크 I/O 비용도 많이 들고, 또 그 전체 데이터를 처리하는 비용도 많이 들기 때문에 수행시간이 높게 나온다. FS_V의 경우 매 단계마다 데이터베이스를 필터링해서 그 사이즈를 줄이고, 다음 단계에서 사이즈가 축소된 데이터베이스를 스캔하기 때문에 디스크 I/O의 비용도 줄어들고, 그 처리 비용도 줄어들게 되어 시퀀스의 길이가 길어지면서 그 수행시간이 줄어든다. SPADE_V와 AVL은 k = 3부터 메인 메모리 수행을 한다. SPADE_V의 경우 k = 1, k = 2일 때 너무 복잡한 변환 과정으로 인해 그 처리해야 하는 오버헤드가 많이 들고, 그러므로 그 수행 시간이 높게 나온다. AVL의 경우 전반적으로 성능이 가장 우수하다.



[그림 4.4] 시퀀스 길이에 따른 각 알고리즘의 실행 시간

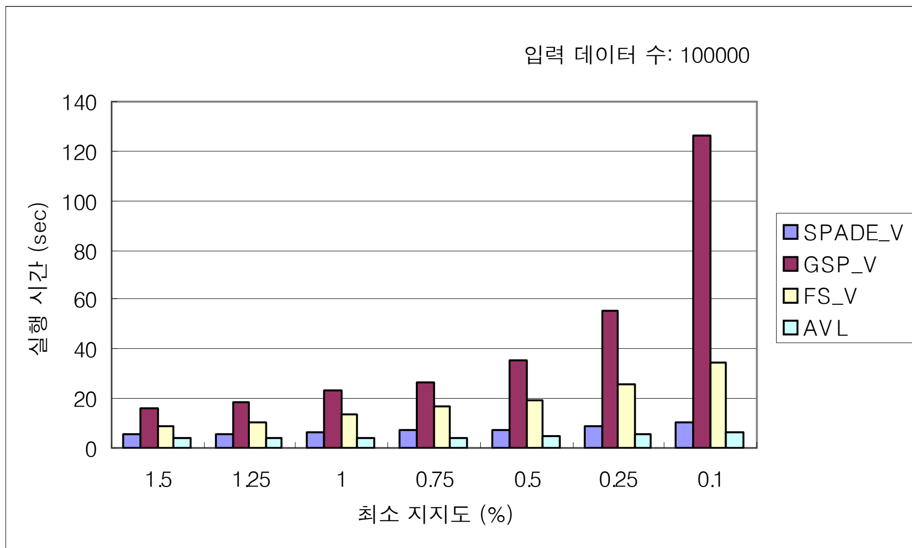
③ 최소 지지도에 따른 각 알고리즘의 실행 시간

그림 4.5의 그래프를 보면 GSP_V의 경우 최소 지지도가 낮아질수록 그 실행 시간이 다른 알고리즘에 비해 상당히 높음을 알 수 있다. 이는 그림 4.6에서도 볼 수 있듯이 최소 지지도가 낮아질수록 탐사되는 빈발 시퀀스의 개수도 많아지고, 그 탐사되는 시퀀스의 길이도 길어진다. 탐사된 시퀀스의 길이가 길어진다는 것은 그만큼 전체 트랜잭션 데이터베이스를 반복적으로 더 스캔해야 하며, 그에 따른 디스크 I/O와 그 데이터를 처리해야 하는 오버헤드가 더 부과된다는 것을 의미한다.

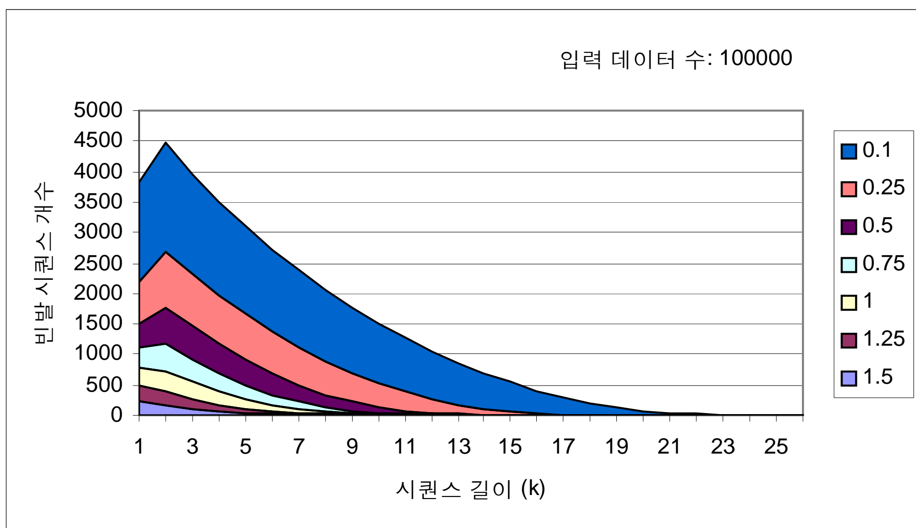
FS_V의 경우도 최소 지지도가 낮아질수록 그 실행 시간이 증가되는 것

을 볼 수 있는데, GSP_V에 비해 그 증가폭이 상당히 낮음을 알 수 있다. 이는 FS_V 알고리즘의 특성상 시퀀스의 길이가 길어지면서 데이터베이스의 사이즈를 축소시키고, 다음 단계에서 축소된 데이터베이스를 스캔하기 때문에 전체 데이터베이스를 반복적으로 스캔하는 GSP_V보다 그 실행 시간이 적게 걸리는 것을 의미한다.

그리고 SPADE_V와 AVL의 경우는 최소 지지도가 낮아질수록 그 실행시간은 분별하기 어려울 정도로 아주 조금 증가한다. SPADE_V의 경우 최소 지지도가 0.25%일 때 그 실행시간은 8.7031초이고, 최소지지도가 0.1%일 때는 10.1922초 수행된다. AVL의 경우도 마찬가지인데, 이 알고리즘의 경우 최소 지지도가 0.25%일 때 그 실행시간은 5.2688초이고, 최소지지도 0.1%일 때는 6.2015초 걸린다. 최소 지지도가 낮아져서 탐사되는 빈발 시퀀스의 수가 증가하고 탐사되는 시퀀스의 길이가 길어진다 하더라도 이 알고리즘들은 k = 3일 때부터 메인 메모리에서 수행을 하기 때문에 k = 1, k = 2일 때 큰 오버헤드가 없다면 그 수행 시간의 변화는 크지 않을 것으로 판단된다.



[그림 4.5] 최소 지지도에 따른 각 알고리즘의 실행 시간



[그림 4.6] 시퀀스 길이에 따른 각 최소 지지도별 탐사된 빈발 시퀀스

④ FS_V 알고리즘의 비트 연산 vs. 덧셈 연산

FS_V 알고리즘은 시퀀스의 길이가 증가할 때마다 반복적으로 트랜잭션 데이터베이스를 스캔하여 후보 시퀀스의 지지도를 계산한 후, k = 2일 때부터 다음 단계에 사용될 시퀀스만 필터링하여 데이터베이스에 기록한다. 표 4.3과 표 4.4에서 볼 수 있듯이 시퀀스의 길이가 증가할수록 데이터베이스의 크기는 점점 더 줄어들고, 그에 따른 수행시간도 점점 줄어드는 것을 알 수 있다. k가 증가할 때마다 트랜잭션 데이터베이스의 크기를 점증적으로 줄이는데, 그 방법으로 비트 연산이나 덧셈 연산을 이용한다. 비트 연산의 경우 덧셈 연산보다 그 계산이 더 간단하여 알고리즘 전반적인 수행시간이 더 적게 걸릴 것으로 예상되지만, 다음 단계에 사용될 시퀀스를 필터링하는 부분에서 더 적에 필터링이 되기 때문에 덧셈 연산보다 그 수행시간이 빠르지 않다. 오히려 덧셈 연산이 비트 연산보다 필터링 기능이 더 우수하여 그 만큼 데이터베이스의 사이즈가 더 적어지고, 그 데이터를 처리하는 시간이 더 적게 걸린다.

DB*k*	비트 연산	덧셈 연산
DB0	12,991	12,991
DB2	7,225	7,225
DB3	5,343	5,285
DB4	4,469	4,361
DB5	3,531	3,431
DB6	2,610	2,498
DB7	1,780	1,703
DB8	1,092	1,063
DB9	646	646
DB10	387	388
DB11	329	329
DB12	219	219
DB13	0	0

[표 4.3] FS_V 알고리즘의 시퀀스 길이에 따른 데이터베이스 크기

Fk	비트연산	덧셈연산
F1	1.4829	1.4875
F2	3.197	3.1561
F3	2.1983	2.1314
F4	1.7235	1.6592
F5	1.3857	1.3281
F6	1.0628	0.986
F7	0.7405	0.686
F8	0.4751	0.4422
F9	0.2828	0.2703
F10	0.1688	0.1625
F11	0.125	0.1189
F12	0.0905	0.089
F13	0.0265	0.0266
실행시간	12.9594	12.55

[표 4.4] FS_V 알고리즘의 시퀀스 길이에 따른 실행 시간

Ⅴ. 결론

본 논문에서는 순회 패턴 탐사 문제를 해결하기 위해 기존의 알고리즘들을 순회 패턴 탐사 문제에 맞게 변형하여 그 문제를 해결하고자 했다. 즉, 객체들이 서로 연결되어 있고, 객체들 사이의 접근은 일정한 방향성을 갖는 그러한 환경 하에서 객체들을 순차적으로 접근하는 일정한 패턴을 찾는 문제를 해결하려 했으며, 이를 위해 기존의 알고리즘인 SPADE, GSP, FS를 변형한 SPADE_V, GSP_V, FS_V를 제안했고, 여기에 새로운 배열 자료구조와 수직 식별자 데이터베이스 개념을 이용한 AVL을 제안했다. 위의 각 알고리즘에 대해 그 성능을 평가하기 위해 실세계 데이터인 교통 카드 트랜잭션 데이터베이스를 적용해 보았고, 그 결과를 토대로 각 알고리즘을 비교 분석해 보았다.

SPADE_V 알고리즘은 수직 데이터베이스 개념을 이용한다. F_1을 탐사할 때 수직 데이터베이스를 한번만 스캔하고, 그 다음부터는 메인 메모리 수행을 한다. 이는 수직 데이터베이스가 각 시퀀스에 대한 위치 정보를 유지하기 때문에 가능하다. 이 알고리즘은 데이터베이스의 사이즈가 작은 경우에 가능하다. 그리고 F_1과 F_2 단계에서 너무 복잡한 변환 과정을 수행하므로 그 성능 병목 현상을 일으킨다. AVL 알고리즘은 F_1을 탐사할 때 해시 기법을 이용하고, F_2 단계에서 배열 구조를 이용하여 그 수행을 개선하였으며, F_3 단계부터는 수직 데이터베이스 개념을 이용한다. F_1과 F_2의 각 단계에서 트랜잭션 데이터베이스를 스캔하고, 그 다음 단계부터는 주 메모리 수행을 한다. 대부분의 경우 F_2 단계까지 구해지면 처리해야 하는 데이

터의 수가 상당히 줄어들기 때문에 그 다음단계부터는 메인 메모리에서 수행이 가능하다. 만약 데이터베이스의 사이즈가 아주 크다면, 메인 메모리에서 수행할 수 있는 단계까지 트랜잭션 데이터베이스를 스캔하면서 구하고, 메인 메모리에 충분히 수용할 수 있는 단계에 수직 데이터베이스 개념을 적용한다. 대부분의 순회 패턴 탐사나 연관 규칙 탐사 등이 그러하듯이 F_1과 F_2를 구하는 부분에서 알고리즘 전체 수행시간의 병목현상을 갖는다. 그러나 AVL은 다른 알고리즘에 비해 각 단계에서 더 나은 성능을 보인다.

GSP_V는 시퀀스의 길이가 증가할 때마다 전체 트랜잭션 데이터베이스를 반복적으로 스캔한다. 그 만큼 디스크 I/O 비용과 그 데이터를 처리해야하는 비용이 더 들기 때문에 다른 알고리즘에 비해 그 성능이 떨어진다. 특히 탐사될 시퀀스의 길이가 길어질수록 그만큼 데이터베이스를 반복 스캔해야하기 때문에 성능이 더 떨어진다. FS_V 알고리즘은 F_1을 제외한 모든 단계에서 데이터베이스를 필터링해서 점증적으로 데이터베이스의 사이즈를 줄인다. 그래서 다음 단계에 사용할 데이터베이스는 이전 단계에서 필터링 되어 사이즈가 축소된 데이터베이스를 사용하므로 그만큼 디스크 I/O 비용이나 그 데이터 처리비용이 적기 때문에 GSP_V보다 우수하다. FS_V 알고리즘은 수행 중에 사이즈가 축소된 데이터베이스를 생성하므로 중간 단계 데이터베이스를 저장할 공간이 필요하다. GSP_V나 FS_V는 입력 트랜잭션 데이터베이스가 아주 클 경우에 유용하며, 전반적으로 GSP_V보다 FS_V의 성능이 우수하다.

본 논문에서 구현한 순회 패턴 탐사 알고리즘들의 실험 결과 AVL 알고리즘의 성능이 가장 우수했음을 알 수 있었고, 그 확장성은 입력데이터의 수에 선형적으로 증가함을 볼 수 있었다. 지금까지 교통 카드 트랜잭션 데이터베이스에 순회 패턴 탐사 알고리즘을 적용시켜 보았고, 향후 탐사된 패

턴을 좀 더 효과적으로 응용하고 분석할 수 있는 사용자 인터페이스 개발이 병행 연구되어야 할 것이다.

참고 문헌

[1] Mohammed J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences", Machine Learning Journal, pp. 31-60, Vol 42, No 1-2, 2001.

[2] R. Srikant and R. Agrawal: "Mining Sequential Patterns: Generalizations and Performance Improvements", In Proc. of the 5th International Conference on Extending Database Technology, Avignon, France, March 1996.

[3] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M-C. Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth", In Proc. 2001 Int. Conf. on Data Engineering (ICDE'01), pp. 215-224, Heidelberg, Germany, April 2001.

[4] M.-S. Chen, J. S. Park, and P. S. Yu, "Efficient Data Mining for Path Traversal Patterns", IEEE Trans. on Knowledge and Data Engineering, pp. 209-221, Vol 10, No 2, 1998.

[5] 박종수, 하미라, 김연호. "웹 로그 파일에서 순회 패턴 탐사 알고리즘", 데이

터마이닝 WORKSHOP, 2001.

[6] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining", In Proc. 2000 Int. Conf. Knowledge Discovery and Data Mining (KDD'00), pp. 355-359, Boston, MA, August 2000.

[7] R. Agrawal and R. Srikant, "Mining Sequenctial Patterns", In Proc. Int. Conf. Data Engineering (ICDE'95), pp. 3-14, Taipei, Taiwan, March 1995.

[8] J. S. Park, M. S. Chen, and P. S. Yu, "Using a Hash-Based Method with Transaction Trimming for Mining Association Rules", IEEE Trans. on Knowledge and Data Engineering, Vol 9, No. 5, pp. 813-825, September 1997.

[9] H. Mannila, H. Toivonen, and A. I. Verkamo. "Discovery of frequent episodes in event sequences", Data Mining and Knowledge Discovery, Vol. 1, No. 3, pp. 259-289, 1997.

[10] M. S. Chen, J. S. Park, and P. S. Yu, "Data Mining for Path Traversal Patterns in a Web Environment", In Proc. of the 16th Int. Conf. on Distributed Computing Systems, pp. 385-392, Hong Kong, May 1996.

[11] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules in large database", In Proc. of ACM SIGMOD Conference on Management of Data, Washington D.C., pp. 207-216, May 1993.

[12] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", In Proc. of the 20th Int. Conf. on Very Large Databases, Santiago, Chile, September 1994.

[13] J. S. Park, M.-S. Chen, and P.S. Yu, "An Effective Hash-Based Algorithm for Mining Association Rules", In Proc. of ACM SIGMOD Int. Conf. on Management of Data, pp. 175-186, San Jose, California, May 1995.

[14] J. Srivastava, R. Cooley, M. Deshpande, P-T Tan, "Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data", SIGKDD Explorations, Vol. 1, No. 2, pp.12-23, 2000.

[15] R. Cooley, B. Mobasher, and J. Srivatiava, "Data Preparation for Mining World Wide Web Browsing Patterns", Knowledge and Information Systems, Vol. 1, No. 1, pp. 5-32, Feb. 1999.

[16] J. Pei, J. Han. B. Mortazavi-Asl, and H. Zhu, "Mining Access Patterns Efficiently from Web Logs", In Proc. of the 4th Pacific-Asia Conference on Knowledge Discovery and Data mining,

pp. 396-407, 2000.

[17] E.G. Coffman Jr. and J. Eve, "File structures using hashing functions", Communications of the ACM, Vol. 13, No. 7, pp. 427-432, July 1970.

[18] K. S. Lee and J. S. Park, "Traversal Pattern Analysis of Transit Users in The Metropolitan Seoul", In Proc. of International Forum on the Public Transportation Reform in Seoul, 6 pages, July 7-8, 2005.

ABSTRACT

Comparison and Analysis of Algorithms for Mining Traversal Patterns in a Transaction Database by Transportation Cards

Choi, Kyesook
Department of Computer Science
Graduated School of
Sungshin Women's University

The major reason that data mining has attracted a great deal of attention in the information industry in recent years is due to the wide availability of huge amounts of data and the imminent need for turning such data into useful information and knowledge. Traversal pattern mining is an important data mining problem with broad applications.

In this research, we examine the issue of mining traversal patterns among items in a transaction database by transportation cards, where each transaction consists of customer-id, transaction-id, and a list of the items such as stations between origins and destinations in a transaction. We derived algorithms to find out the traversal patterns. We implemented the algorithms and evaluated the performance.