DOI: 10.3745/KIPSTD.2011.18D.2.081

맵리듀스 프레임웍 상에서 맵리듀스 함수 호출을 최적화하는 순차 패턴 마이닝 기법

김 진 현[†] · 심 규 석[††]

요 약

시퀀스(sequence) 데이터가 주어졌을 때 그 중에서 빈번(frequent)한 순차 패턴을 찾는 순차 패턴 마이닝(sequential pattern mining)은 여러 어플리케이션(application)에 사용되는 중요한 데이터마이닝 문제이다. 순차 패턴 마이닝은 웹 접속 패턴, 고객 구매 패턴, 특정 질병의 DNA 시퀀스를 찾는 등 광범위한 분야에서 사용된다.

본 논문에서는 맵리듀스(MapReduce) 프레임웍 상에서 맵리듀스 함수 호출을 최적화하는 순차 패턴 마이닝 알고리즘을 개발하였다. 이 알고리즘은 여러 대의 기계에 데이터들을 분산시켜 병렬적으로 빈번한 순차 패턴을 찾는다. 실험적으로 다양한 데이터를 이용하여 파라미터 값을 변화시켜가며 제안된 알고리즘의 성능을 종합적으로 확인하였다. 그리고 실험 결과를 통해 제안된 알고리즘은 기계 수에 대해 선형적인 속도 개선을 보인다는 것을 확인하였다.

키워드 : 데이터마이닝, 순차 패턴 마이닝, 맵리듀스, 하둡, 병렬 처리

Sequential Pattern Mining with Optimization Calling MapReduce Function on MapReduce Framework

Jinhyun Kim[†] · Kyuseok Shim[††]

ABSTRACT

Sequential pattern mining that determines frequent patterns appearing in a given set of sequences is an important data mining problem with broad applications. For example, sequential pattern mining can find the web access patterns, customer's purchase patterns and DNA sequences related with specific disease.

In this paper, we develop the sequential pattern mining algorithms using MapReduce framework. Our algorithms distribute input data to several machines and find frequent sequential patterns in parallel. With synthetic data sets, we did a comprehensive performance study with varying various parameters. Our experimental results show that linear speed up can be achieved through our algorithms with increasing the number of used machines.

Keywords : Data Mining, Sequential Pattern Mining, MapReduce, Hadoop, Parallel Processing

1. 서 론

IT기술과 컴퓨터 및 인터넷산업이 발달하면서 누적된 정보의 양이 늘어나고 있고, 기업과 개인이 목적에 의해 이 정보들을 분석할 필요성이 높아짐에 따라 데이터들을 효과적으로 처리, 분석하고 그 속에서 유용한 정보들을 찾아내는 데이터마이닝(data mining)이 중요한 분야가 되었다.

이 논문에서는 데이터마이닝의 가장 중요한 기법 중 하나인 순차 패턴 마이닝에 대해 다룰 것이다. 순차 패턴 마이닝이란 시퀀스(sequence) 데이터 베이스가 주어졌을 때 그 중에서 빈번한 순차 패턴을 찾아 주는 데이터마이닝 문제이다. 이 순차 패턴 마이닝은 고객의 물품 구매 습관, 웹 접근 패턴, 실험 결과, 질병 치료, 자연 재해, DNA 시퀀스를 분석하는 등 여러 분야에 적용된다.

순차 패턴 마이닝 기법은 그 특성상 데이터베이스의 크기가 커지면 많은 양의 연산이 필요하게 되므로 이를 위한 속도 개선 알고리즘들이 많이 개발되어 왔다. 하지만 데이터베이스의 크기가 매우 큰 경우 하나의 기계에서 순차 패턴 마이닝을 실행하면 오랜 시간이 걸리게 되어 응용(application)

※ 이 논문은 2011년 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2011-0000349).
† 준 회 원 : 서울대학교 전기컴퓨터공학부 박사과정
†† 정 회 원 : 서울대학교 전기컴퓨터공학부 교수
논문접수 : 2010년 9월 27일
수 정 일 : 1차 2011년 1월 24일, 2차 2011년 2월 11일
심사완료 : 2011년 2월 14일

Sequence_id	Sequence
10	$<(a)(abc)(ac)(d)(cf)>$
20	$<(ad)(c)(bc)(ae)>$
30	$<(ef)(ab)(df)(c)(b)>$
40	$<(e)(g)(af)(c)(b)(c)>$

(그림 1) 시퀀스 데이터베이스 D

프로그램에 바로 적용하기 어렵다. 이를 해결하기 위해 여러 대의 기계에서 병렬적으로 작업을 처리하는 분산 처리 시스템이 필요하게 되었다.

최근 대용량의 인터넷 데이터를 효율적으로 관리하기 위해서 분산 파일 시스템이 도입되고 있다. 구글에서는 구글 파일시스템(Google file system)과 빅테이블(Big-Table)을 개발하여 사용하고 있으며 효과적인 분산 프로그래밍 환경을 지원하기 위해 맵리듀스(MapReduce[1]) 프레임워을 적용하고 있다. 이러한 추세에 맞춰 오픈 소스 진영에서도 맵리듀스를 구현한 하둡(Hadoop[2])이라는 소프트웨어 플랫폼이 개발되었다. 하둡은 하둡 분산 파일 시스템(Hadoop Distributed File System: HDFS)을 저장 공간으로 제공하고, 맵리듀스를 분산 프로그래밍 구현 방법으로 사용한다. 맵리듀스는 개별적인 데이터를 분석해서 그와 연관된 중간 데이터를 생산하는 맵(Map) 함수 과정과, 이 중간데이터들을 모아서 종합된 결과를 얻는 리듀스(Reduce) 함수 과정으로 구성된다. 맵리듀스는 기본적으로 여러 대의 기계에서 동시에 연산을 진행할 수 있기 때문에 연산 비용이 높은 작업에 속도향상을 가져올 수 있다.

본 논문에서는 클라우드 컴퓨팅 상의 맵리듀스(MapReduce) 프레임웍을 이용한 순차 패턴 마이닝 알고리즘을 개발하였다. 맵리듀스 프레임웍을 통해 데이터 연산을 분산 처리하여 순차 패턴 마이닝 알고리즘의 수행 속도를 현저히 빠르게 하였다.

본 논문의 구성은 다음과 같다. 2장에서는 논문의 이해를 돕기 위한 기본 개념을 좀 더 구체적으로 설명하고 순차 패턴 문제를 정의하며 이를 해결하기 위해 본 논문에서 사용하는 프리픽스스팬과 맵리듀스 프레임웍을 소개한다. 3장에서는 기존 순차 패턴 마이닝에 대해서 진행되어 온 관련 연구들에 대해서 알아보고 4장에서는 맵리듀스 프레임웍을 이용한 순차 패턴 마이닝 알고리즘을 제안한다. 5장에서는 제안한 알고리즘을 패턴의 개수나 최소 지지도와 같은 다양한 인자들을 바꿔가며 시행한 실험 결과를 통해 성능의 효율성을 보여준다. 마지막으로, 6장에서는 결론과 향후 과제를 제시한다.

2. 기본 개념

2.1 순차 패턴

물품 구매 내역 시퀀스 데이터베이스가 주어졌다고 하자. 각각의 시퀀스는 한 사용자가 구매한 물품 내역이 되는데 이 때 시퀀스는 아이템의 집합을 순서가 있게 나열한 것을 뜻한다. 아이템집합 $I = \{i_1 \ldots i_M\}$는 데이터베이스의 모든 아이템의 집합이며 각 시퀀스 s는 $<s_1 \ldots s_n>$으로 표현할 있다. 이 때 각각의 s_i는 I의 부분집합이 된다. 즉 각 s_i는 이템의 집합이 되고 시퀀스의 원소(element)라고 한다. 그고 아이템들에는 순서가 있어 한 원소 안에서 아이템들 그 순서에 따라 나열된다. 또한 시퀀스의 길이는 시퀀스 구성하는 아이템의 개수로 정의된다.

정의 2.1.1 시퀀스 $a = <a_1 \ldots a_n>$와 $b = <b_1 \ldots b_m>$ 있을 때, $a_1 \subseteq b_{i_1}, \ldots, a_n \subseteq b_{i_n}$를 만족시키는 $1 \le i_1 < \ldots < i_n \le$ 인 정수 $i_1 \ldots i_n$ 들이 존재하면 b는 a를 지지한다고 하고 는 b의 부분시퀀스(sebsequence)라고 한다.

정의 2.1.2 시퀀스 데이터베이스 S 내에서 특정 시퀀스를 지지하는 시퀀스의 비율을 a에 대한 지지도(support) 한다. 또 최소 지지도 min_sup이 주어졌을 때 어떤 시퀀 a에 대한 지지도가 min_sup 이상이라면 a는 최소 지지도 만족한다고 한다.

순차 패턴 문제 정의: 시퀀스 데이터베이스와 최소 지지도가 주어졌을 때 주어진 데이터베이스 내에서 최소 지지도를 만족하는 모든 시퀀스를 찾는 것을 순차 패턴 마이닝 문제라고 한다[3]. 이 때 여기에서 찾아진 시퀀스들을 빈번한 순차 패턴이라고 한다.

2.2 프리픽스스팬

이 절에서는 순차 패턴 문제를 해결하는 알고리즘인 프리픽스스팬(PrefixSpan)[4]에 대해 설명한다.

정의 2.2.1 시퀀스 $\alpha=<e_1 \ldots e_n>$, $\beta=<e'_1 \ldots e'_m>(m \le n$ 에 대해 β가 α의 접두사(prefix)라면 (1) $1 \le i \le m-1$인 모든 i에 대해 $e'_i=e_i$ 를 만족하고 (2) $e'_m \subseteq e_m$이며 (3) $(e_m-e'_m)$에 속한 모든 아이템이 e'_m에 속한 모든 아이템들보다 순서상 뒤에 위치한다. 이때 α는 접두사 β를 가진다고 한다.

정의 2.2.2 시퀀스 α의 부분시퀀스 α′이 있을 때 α′이 접두사 β에 대한 α의 프로젝션(projection)이라면 (1) β가 α′의 접두사이고 (2) 접두사 β를 가지는 α의 모든 부분시퀀스는 α′의 부분 시퀀스이다. β가 α의 부분시퀀스가 아니라면 β에 대한 α의 프로젝션은 빈 시퀀스이다.

정의 2.2.3 $\alpha'=<e_1 \ldots e_n>$가 접두사 $\beta=<e_1 \ldots e_{m-1} e'_m>(m \le n)$에 대한 α의 프로젝션일 때 $\gamma=<e''_m e_{m+1} \ldots e_n>$를 접두사 β에 대한 α의 접미사(postfix)라고 한다. 여기서 $e''_m=(e_m-e'_m)$이다.[1] γ가 접두사 β에 대한 α의 접미사인 경우 $\gamma=\alpha/\beta$나 $\alpha=\beta \cdot \gamma$로 표시한다.

정의 2.2.4 시퀀스 α에 대한 시퀀스 데이터베이스 S의 프로젝션된 데이터베이스는 $S|_\alpha$라고 표기하며 α에 대한 S의 시퀀스들의 접미사들의 집합으로 정의된다. $S|_\alpha$를 구하는 것을 S를 α에 대해 프로젝션(projection)한다고 한다.

이제 프리픽스스팬의 수행 방식을 설명하도록 하겠다. 프리픽스스팬은 재귀적으로 수행되며 입력으로 길의 L의 접두사 α와 시퀀스 데이터베이스 S의 α에 대한 프로젝션된 데

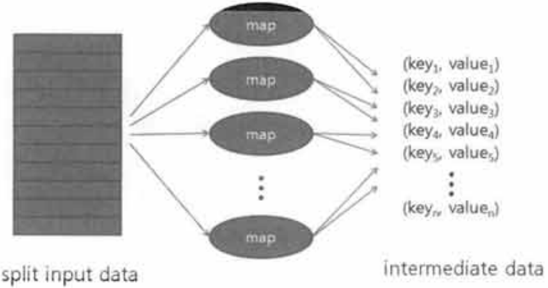
1) 만약 e_m이 공집합이 아니면, 접미사 γ를 $<(_e''_m$의 아이템들$)\ e_{m+1} \cdots e_n>$으로 표현하기도 한다. 즉 접미사의 첫 원소에 _가 있는 경우 이 원소의 아이템은 접두사의 마지막 원소와 프로젝션에서는 같은 원소에 속했다는 것을 뜻한다.

Prefix	Projected database	Sequential Patterns
<(a)>	<(abc)(ac)(d)(cf)>, <(_d)(c)(bc)(ae)>, <(_b)(df)(c)(b)>, <(_f)(c)(b)(c)>	<(a)>, <(a)(a)>, <(a)(b)>, <(a)(bc)>, <(a)(bc)(a)>, <(a)(b)(a)>, <(a)(b)(c)>, <(ab)>, <(ab)(c)>, <(ab)(d)>, <(ab)(f)>, <(ab)(d)(c)>, <(a)(c)>, <(a)(c)(a)>, <(a)(c)(b)>, <(a)(c)(c)>, <(a)(d)>, <(a)(d)(c)>, <(a)(f)>
<(b)>	<(_c)(ac)(d)(cf)>, <(_c)(ae)>, <(df)(c)(b)>, <(c)>	<(b)>, <(b)(a)>, <(b)(c)>, <(bc)>, <(bc)(a)>, <(b)(d)>, <(b)(d)(c)>, <(b)(f)>
<(c)>	<(ac)(d)(cf)>, <(bc)(ae)>, <(b)>, <(b)(c)>	<(c)>, <(c)(a)>, <(c)(b)>, <(c)(c)>
<(d)>	<(cf)>, <(c)(bc)(ae)>, <(_f)(c)(b)>	<(d)>, <(d)(b)>, <(d)(c)>, <(d)(c)(b)>
<(e)>	<(_f)(ab)(df)(c)(b)>, <(af)(c)(b)(c)>	<(e)>, <(e)(d)>, <(e)(a)(b)>, <(e)(a)(c)>, <(e)(a)(c)(b)>, <(e)(b)>, <(e)(b)(c)>, <(e)(c)>, <(e)(c)(b)>, <(e)(f)>, <(e)(f)(b)>, <(e)(f)(c)>, <(e)(f)(c)(b)>
<(f)>	<(ab)(df)(c)(b)>, <(c)(b)(c)>	<(f)>, <(f)(b)>, <(f)(b)(c)>, <(f)(c)>, <(f)(c)(b)>

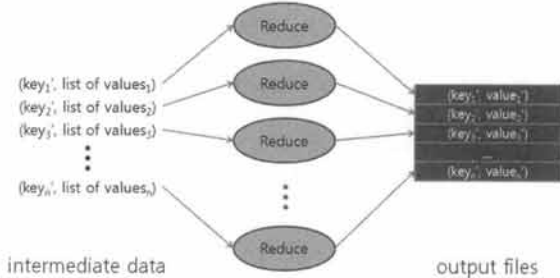
(그림 2) 빈번한 순차 패턴

이터베이스 $S|_\alpha$를 받아 α에서 확장된 길이 L+1 이상의 모든 순차 패턴을 구한다. 따라서 길이가 0인 빈 시퀀스와 시퀀스 데이터베이스 S를 입력으로 수행하면 S에서의 모든 순차 패턴을 찾는다. 프리픽스스팬이 수행되면 우선 입력 $S|_\alpha$를 읽어 최소 지지도 이상의 시퀀스에서 등장하는 빈번한 아이템을 찾는다. 각각의 빈번한 아이템들을 접두사 α에 덧붙여 확장시킨 새로운 접두사들은 길이 L+1의 빈번한 순차 패턴이 된다. 새로운 접두사들과 이 새로운 접두사에 대한 $S|_\alpha$의 프로젝션된 데이터베이스를 입력으로 각각 재귀적으로 프리픽스스팬을 수행하면 α에서 확장되는 모든 빈번한 순차 패턴을 찾을 수 있다. 자세한 설명은 [4]에서 확인할 수 있다.

예제 2.2.1 (그림 1)의 데이터베이스 D에서 최소 지지도 50%를 만족하는 순차 패턴을 찾는 문제를 생각해 보자. 데이터베이스를 구성하는 시퀀스의 개수는 4이므로 최소 지지도를 만족하려면 2개 이상의 시퀀스에서 등장해야 한다. D에서의 모든 빈번한 순차 패턴을 찾기 위해 빈 시퀀스 <>와 D를 입력으로 프리픽스스팬을 수행한다. D에서 빈번한 아이템 {a, b, c, d, e, f}을 찾고 이들을 <>에 덧붙인 <(a)>, <(b)>, <(c)>, <(d)>, <(e)>, <(f)> 를 길이 1의 빈번한 순차 패턴으로 찾을 수 있다. 이제 이 각각의 접두사들에 대한 D의 프로젝션된 데이터베이스를 만들고 재귀적으로 프리픽스스팬을 수행한다. $D|_{<(a)>}$는 (그림 2)에서와 같이 4개의 프로젝션으로 구성된 데이터베이스가 되며 <(a)>와 $D|_{<(a)>}$를 입력으로 프리픽스스팬을 수행하면 $D|_{<(a)>}$에서 빈번한 아이템을 찾아 이를 현재 접두사 <(a)>의 뒤에 붙여 길이 2의 빈번한 순차 패턴 <(a)(a)>, <(a)(b)>, <(a)(c)>, <(a)(d)>, <(a)(e)>, <(a)(f)>, <(a b)>을 찾는다. 이에 대한 프로젝션된 데이터베이스들을 사용해 재귀적으로 프리픽스스팬을 수행하고 더 이상 수행할 프로젝션된 데이터베이스가 없을 때까지 이 과정을 반복하면 모든 빈번한 순차 패턴을 찾을 수 있다. 찾아진 모든 빈번한 순차 패턴은 (그림 2)에 나타냈다.



(그림 3) 맵 함수의 데이터 흐름



(그림 4) 리듀스 함수의 데이터 흐름

2.3. 맵리듀스

맵리듀스는 대용량 데이터를 처리하기 위해 개발된 프로그래밍 모델로 개별적인 데이터를 분석해서 그와 연관된 중간데이터를 생산하는 맵(Map) 함수 과정과, 이 중간데이터들을 모아서 종합된 결과를 얻는 리듀스(Reduce) 함수 과정으로 구성된다[1].

맵과 리듀스 함수는 데이터를 처리하고 통합하기 위해 키(key)와 값(value)이라는 개념을 사용한다. 키는 어떤 데이터에 대한 고유한 식별 값을 나타내고, 값은 이 식별 값에 대한 각각의 데이터의 해당 값을 나타낸다.

맵 함수는 (그림 3)과 같이 입력 파일들을 줄 단위로 읽는다. 입력은 각 맵 함수로 분산되며 연산을 거쳐 각 데이터에 대해 맵 함수가 생성한 (키, 값) 쌍을 방출(emit)한다. 맵 함수에서 방출된 (키, 값) 쌍들은 같은 키에 대해 모여 (키, 값 리스트) 쌍의 형태로 저장된다. 각 로컬 디스크에서 방출된 (키, 값 리스트) 쌍들은 리듀스 함수가 시작되기에 앞서 전체 맵리듀스 프레임워크 상에서 키 값에 대해 정렬하여 같은 키를 가지는 값 리스트로 또 한번 통합된다. 이들은 각각의 리듀스 함수의 입력으로 나뉘어져 들어가 분산처리된다(그림 4). 이 때 맵리듀스 프레임워크은 각 키에 대한 해시 함수(hash function)를 이용해 (키, 값 리스트)를 리듀

스 함수에 분산시킨다. 그 결과 (키, 값 리스트)들이 여러 리듀스 함수로 분산되며, 같은 키를 가지는 것들은 같은 리듀스 함수로 들어간다. 리듀스 함수로 들어간 (키, 값 리스트)쌍은 사용자가 정의한 연산 과정을 거쳐 새로운 식별자 키를 만들고, 그에 따른 또 다른 값을 만들어 (키, 값) 쌍을 방출한다. 맵리듀스는 사용자들이 맵 함수와 리듀스 함수를 잘 설계함으로써 여러 가지 응용에 대해 분산 시스템 환경을 지원한다[1].

```
Function SP_NAIVE.main(S, min_sup)
Input A sequence database S, Minimum support threshold
min_sup
Output The complete set of frequent sequential patterns
begin
1. L := 0, DB := S, allPat := { }
2. while DB is not empty do {
3.   Set DB and min_sup to Count.Map
4.   runMapReduce(Count.Map, Count.Reduce)
5.   freqPat := getReduceOutput()
6.   Set DB and freqPat to Proj.Map
7.   runMapReduce(Proj.Map, Proj.Reduce)
8.   projDB := getReduceOutput()
9.   allPat := allPat ∪ freqPat
10.  DB := projDB }
11.  return allPat
end
```

```
Function Count.Map(key, value)
key null
value a string
begin
1. (prefix, postfix) = preProcess(value), EmitList := { }
2. for each element e in postfix do {
3.   itemList := { }
4.   for each item i in e do {
5.     if (prefix • i ∉ EmitList) then {
6.       emit(prefix • i, 1)
7.       itemList = itemList∪prefix • i }
8.     if the last element of prefix ⊆ itemList and
       (prefix • _i∉emitList) then {
9.       emit(prefix • _i, 1)
10.      emitList := emitList ∪ prefix • _i }
11.    itemList := itemList ∪ i } }
end
```

```
Function Count.Reduce(key, valuelist)
key a prefix
value a list of 1s
begin
1. count := 0
2. for each value v in valuelist do {
3.   count = count + 1
4.   If count ≥ min_sup then {
5.     emit(key, null)
6.     break } }
end
```

(그림 5) SP_NAIVE.Main과 SP_NAIVE.Count의 Map과 Reduce

```
Function Proj.Map(key, value)
key prefix
value a string
begin
1. Load freqPat, EmitList := { }
2. (prefix, postfix) = preProcess(value)
3. for each element e in postfix do {
4.   itemList := { }
5.   for each item i in e do {
6.     If (prefix • i∉emitList) and prefix • i∈freqPat then {
7.       emit(prefix • i, value/i)
8.       emitList := emitList ∪ prefix • i }
9.     if the last element of prefix ⊆ itemList and
       (prefix • _i∉emitList) and prefix • _i∈freqPat then {
10.      emit(prefix • _i, value/i)
11.      emitList := emitList ∪ prefix • _i }
12.    itemList := itemList ∪ i } }
end
```

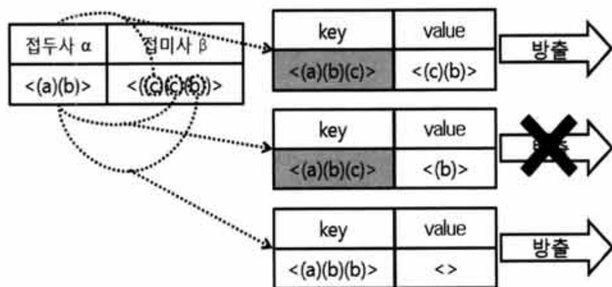
```
Function Proj.Reduce(key, valuelist)
key a prefix
value a list of postfixes
begin
1. count := 0
2. for each value v in valuelist do {
3.   emit(key, v) }
end
```

(그림 6) SP_NAIVE.Proj의 Map과 Reduce

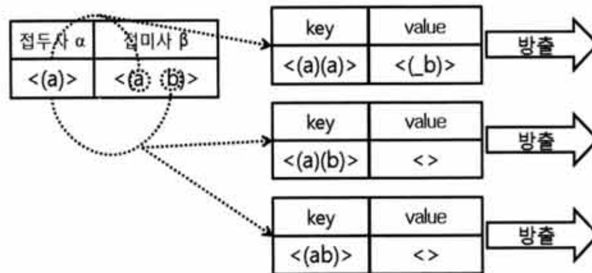
3. 관련 연구

순차 패턴 마이닝 문제는 처음 Agrawal[4]에 의해서 소개되었고 그 후 많은 연구가 진행되어 왔다. 초창기 대부분의 방식은 연관(association) 규칙 마이닝에서 제안된 Apriori 방식[5]이었다. 이에 기초한 GSP[6]는 순차 패턴 마이닝에 후보 생성 검사 접근법을 사용했다. 일단 길이 L의 빈번한 순차 패턴들을 찾았다면 이를 길이 L+1의 빈번한 순차 패턴을 찾는 종자(seed) 집합으로 사용한다. 이 집합을 이용해서 빈번한 순차 패턴이 될 수 있는 후보의 집합을 만든 후 데이터베이스를 읽어 그 중 최소 지지도를 만족하는 빈번한 순차 패턴을 찾는 작업을 수행한다. 이 과정을 새로운 후보 시퀀스가 생성되지 않을 때가지 반복해 모든 순차 패턴을 찾는다.

이 후 Apriori 방식에서 검색 공간을 줄이는 여러 시도가 있었지만 구현 기술이 발전해도 방식 자체의 후보를 생성 확인하는 고유한 비용을 줄일 수는 없었다[3]. 그래서 이 비용을 줄이기 위해 프로젝트에 기초한 방식이 개발되었다. 이 방식은 빈번한 순차 패턴에 대한 프로젝션된 데이터베이스들을 만들어 재귀적으로 순차 패턴을 확장해 나가는 방식이다. 빈번한 순차 패턴에만 아이템을 덧붙여 패턴을 확장하므로 후보를 무작정 많이 만들지 않고 최소 지지도를 확인할 때 프로젝션된 데이터베이스 내에서만 확인하기 때문에 검색 영역이 작아지므로 기존 Apriori 방식의 순차 패턴 마이닝 방식보다 훨씬 좋은 성능을 보인다[3].



(그림 7) 맵 함수에서의 중복 방출



(그림 8) 맵 함수에서의 두 번 방출

한편 순차 패턴 마이닝의 연구 흐름에서는 모든 빈번한 순차 패턴을 찾는 문제를 해결하는 방식뿐만 아니라 닫힌(closed) 순차 패턴을 찾는 문제도 개발되어 왔다[7, 8]. 닫힌 순차 패턴이란 자신을 포함하는 어떤 순차 패턴과도 지지도가 같지 않은 순차 패턴을 뜻한다.

또한 기존의 데이터베이스뿐만 아니라 실시간으로 데이터가 생성되는 스트림(stream) 데이터베이스에서 순차 패턴을 찾는 문제도 연구되고 있다[9, 10].

4. 맵 리듀스 프레임웍 상에서 맵리듀스 함수호출을 최적화하는 순차 패턴 마이닝

4.1 SP_NAIVE 알고리즘

이 장에서는 맵리듀스 프레임웍을 이용해 여러 대의 기계에 프리픽스스팬[7] 방식을 사용하는 SP_NAIVE에 대해 설명한다. 이 알고리즘은 루프를 돌면서 두 단계의 맵리듀스를 반복적으로 수행하도록 구성된다. 각 루프에서는 전 루프에서 찾아진 길이 L의 빈번한 순차 패턴과 그에 대한 프로젝션된 데이터베이스를 통해 길이 L+1의 빈번한 순차 패턴과 이에 대한 프로젝션된 데이터베이스를 찾는 과정을 수행한다. 루프의 첫 번째 맵리듀스에서는 현재 입력 데이터베이스에 대해 빈번한 아이템을 찾아 빈번한 순차 패턴을 만들고 두 번째 맵리듀스에서는 만들어진 패턴에 대한 입력 데이터베이스의 프로젝션된 데이터베이스를 만든다. 생성된 프로젝션된 데이터베이스는 다음 루프의 입력이 되어 이를 반복하며 프로젝션된 데이터베이스가 더 이상 생성되지 않으면 종료한다. (그림 5)와 (그림 6)은 이에 대한 의사 코드이다. 이 때 value는 각 키에 대해 방출되는 값 하나를 뜻하고 valuelist는 이 방출된 값들을 같이 키에 대해 모아서 리스트를 만든 것을 의미한다.

이 알고리즘은 입력으로 시퀀스 데이터베이스 S와 최소 지지도 min_sup을 받게 되고, 수행이 끝나면 모든 빈번한 순차 패턴을 찾는다. DB는 전 루프에서 구해진 빈번한 순차 패턴에 대한 프로젝션된 데이터베이스들의 합집합이 대입되고 allPat에는 모든 루프에서 구해지는 빈번한 순차 패턴이 저장된다. 각 루프에서는 전 루프에서 구한 빈번한 순차 패턴보다 길이가 1 긴 패턴을 구한다. 루프를 살펴보면 DB와 min_sup을 모든 기계에 공유하고 첫 맵리듀스를 호출한다(4행). runMapReduce(Count.Map, Count.Reduce) 함수는 Count.Map함수와 Count.Reduce함수와 함께 맵리듀스를 수행하는 함수이다. Count.Map 함수는 입력 파일을 읽어 우선 preProcessing 함수를 통해 접두사와 접미사를 구분한 뒤(Count.Map의 1행) 이 접두사와 접미사를 변수 prefix와 postfix에 각각 대입한다. 그 후 각 확장된 패턴의 지지도를 계산하기 위해 prefix에 postfix의 각 아이템을 덧붙인 패턴을 키로 하고 숫자 1을 값으로 해 방출하는 역할을 수행한다(6행과 9행). Count.Reduce 함수는 Count.Map 함수에서 방출한 각 키가 최소 지지도를 만족하는지 확인하는 역할을 수행한다. 키의 각 값에 대해 count를 증가시키고 이 값이 최소 지지도가 되면 이 키는 빈번한 패턴이므로 방출한다(Count.Reduce의 4-5행). 수행한 맵리듀스의 결과를 가져오는 getReduceOutput 함수를 통해 freqPat에 현재 루프의 빈번한 패턴을 대입한다(SP_NAIVE.MAIN의 5행). 그 후 Proj.Map함수와 Proj.Reduce함수와 함께 두 번째 맵리듀스 runMapReduce(Proj.Map, Proj.Reduce)를 수행해 구해진 빈번한 순차 패턴에 대한 프로젝션된 데이터베이스들을 만든다. Proj.Map 함수는 Count.Map과 유사한 역할을 수행하는데 모든 패턴에 대해 방출하지 않고 Count.Reduce에서 출력으로 얻어진 빈번한 패턴에 대해서만 방출한다(Proj.Map의 6행). 이 때 값은 각 키에 대한 접미사가 되다(7행). getReduceOutput 함수를 통해 출력을 가져와 DB에 대입하고(SP_NAIVE.Main의 8행) 다음 루프에서 입력으로 사용한다(10행). DB가 공집합이 아닌 한 프로젝션된 데이터베이스가 존재하는 것이므로 루프를 계속 수행한다.

위의 알고리즘의 첫 번째와 두 번째 맵 함수에서 고려해야 할 상황이 있다. (그림 7)과 같이 접미사 β에 같은 아이템이 2번 이상 등장하는 경우 같은 패턴에 대해 여러 번 방출하는 문제가 발생 할 수 있다. β에는 (c)가 두 번 등장해 모든 아이템에 대해 방출하면 키 <(a)(b)(c)>에 대해 값 <(c)(b)>와 <(b)> 두 개의 접미사를 방출한다. 이렇게 되면 실제 이 시퀀스는 키 <(a)(b)(c)>를 한번만 지지하지만 두 번 지지한다고 계산되므로 문제가 발생한다. 이를 방지하기 위해 한 시퀀스에 대해 각 패턴을 이미 방출했는지 확인해야 한다(Count.Map의 5행과 8행).

또 접두사가 현재 방출할 아이템 i가 속한 원소의 부분집합이 되는 경우 이 아이템 i를 접두사의 마지막 아이템집합에 포함시킨 패턴도 이 시퀀스의 부분시퀀스가 되므로 이도 방출해야 할 패턴이 된다. (그림 8)에서 현재 β의(b)를 방출 할 때를 보면 접두사는 <(a)>이고 이는 β의(b)가 속한

```
Function SP-ADVAN.Main(S, min_sup)
Input A sequence database S,
Minimum support threshold min_sup
Output The complete set of frequent sequential patterns
begin
1. DB := S, allPat := { }
2. while DB is not empty do {
3.    Set DB and min_sup to Map
4.    runMapReduce(SP_ADVAN.Map,SP_ADVAN.Reduce)
5.    (freqPat, projDB) := getReduceOutput()
6.    allPat := allPat ∪ freqPat
7.    DB := projDB }
8. return allPat
end
```

(그림 9) SP_ADVAN.Main

```
Function SP-ADVAN.Reduce(key, valuelist)
key a prefix
value a list of postfixes
begin
1. count := 0, preValue := { }, allPat := { }
2. for each value v in valuelist do {
3.    count += 1
4.    preValue := preValue ∪ v
5.    If count  =  min_sup then {
6.      for each element v in preValue do {
7.        emit(key, v) } }
8.    If count ≥ min_sup then {
        emit(key, null) } }
end
```

(그림 10) SP_ADVAN.Reduce

원소의 부분 집합이 되기 때문에 이 (b)는 (_b)로도 해석될 수 있다. 이 경우 키 <(a)(b)>와 <(a b)> 모두에 대해 방출되어야 제대로 된 지지도와 프로젝션된 데이터베이스를 구할 수 있다(Proj.Map의 9행).

이 알고리즘이 정확히 작동하기 위해서는 두 번째 단계의 맵 함수가 첫 번째 맵리듀스에서 찾은 빈번한 패턴을 알고 있어야 하므로 이 맵 함수에 빈번한 패턴을 전달해야 한다. 본 논문에서는 이를 위해 3가지 방식을 제안한다.

첫 번째는 SQL시스템을 이용하는 방식이다. 일단 첫 번째 단계의 리듀스 함수에서 결과로 나온 빈번한 패턴들을 SQL시스템에 입력한다. 예를 들어 첫 번째 리듀스 함수의 결과로 나온 길이 2의 빈번한 패턴이 <(a)(a)>, <(a)(b)>, <(ab)>라면 SQL에서 미리 만들어 놓은 freqPat이란 데이터베이스에 이 값들을 입력(insert)한다. 그 후 두 번째 단계의 맵 함수에서는 만들어진 각 패턴에 대해 SQL시스템에 질의한 후 빈번한 패턴일 경우 방출한다.

두 번째 방식은 하둡의 파일 시스템(HDFS)을 이용해 빈번한 패턴을 공유하는 방법이다. 첫 번째 단계의 리듀스 함수에서 빈번한 패턴들을 하둡 파일 시스템에 기록한다. 위의 예처럼 길이 2의 빈번한 순차 패턴으로 <(a)(a)>, <(a)(b)>, <(a b)>가 나왔다면 이를 freqPat이란 파일에 저장한다. 그리고 두 번째 단계의 맵 함수는 처음에 하둡 파일 시스템를 사용해 이 파일에서 빈번한 패턴을 읽어 메모리에 저장한다. 그리고 이 정보를 이용해 각 패턴이 빈번한지 아닌지 판단해서 방출한다.

마지막 방식은 XML을 이용한 변수로 전달하는 방식이다. 하둡에서는 각 기계에 변수를 전달할 때 XML을 사용한다. 이를 이용해 빈번한 패턴들을 맵 함수들에 공유 할 수 있다. 이 방식도 첫 번째 단계의 리듀스 함수가 빈번한 패턴을 저장하는 것은 앞의 파일 시스템을 이용하는 방식과 같다. 앞에서처럼 <(a)(a)>, <(a)(b)>, <(a b)>가 빈번하다면 첫 번째 단계의 리듀스 함수에서 freqPat이란 파일에 이를 저장한다. 그 후 두 번째 단계의 맵리듀스가 실행되기 전 메인 부분에서 freqPat 파일을 열어 빈번한 패턴을 읽은 후 새로운 변수 strFreqPat을 선언한 후 읽은 패턴을 대입한다. 이는 맵리듀스 프레임웍상에서 XML을 이용해 두 번째 맵리듀스에 변수로 들어가게 되고 두 번째 단계의 맵 함수에서 이를 읽어 빈번한 패턴을 파악한다.

4.2 SP_ADVAN 알고리즘

SP_NAIVE는 한 루프에서 맵리듀스를 2단계 사용하므로 HDFS를 두 번 접근해야 하는 비용과 각 기계들이 빈번한 패턴들을 공유해야 하는 비용이 발생한다. 이를 개선하기 위해 최소 지지도 개수만큼의 프로젝션을 메모리가 가지고 있을 수 있다고 가정하고 맵리듀스를 한 단계만 사용해 빈번한 순차 패턴의 완전 집합을 구하는 알고리즘인 SP_ADVAN을 제안한다(그림 9와 그림 10).

메인 함수는 SP_NAIVE의 메인 함수와 비슷한데 맵리듀스 호출을 한번만 한다는 것을 알 수 있다. 코드를 살펴보면 SP_ADVAN 또한 입력으로 시퀀스 데이터베이스 S와 최소 지지도 min_sup을 받게 되고, 수행이 끝나면 모든 빈번한 순차 패턴의 완전 집합을 얻는다. 1행의 DB는 전 루프에서 구한 빈번한 순차 패턴에 대한 프로젝션된 데이터베이스들의 합집합이고 freqPat에는 현재 루프에서 구해지는 빈번한 순차 패턴이 저장되며 allPat에는 모든 루프에서 구해지는 빈번한 순차 패턴이 저장된다. SP_ADVAN 또한 매 루프에서 전에 구한 빈번한 순차 패턴보다 하나 더 긴 길이의 순차 패턴을 구한다. SP_NAIVE와의 차이는 4행에서 SP_ADVAN.Map 함수와 SP_ADVAN.Reduce 함수와 함께 한번 맵리듀스를 호출해서 최소 지지도를 만족하는 빈번한 순차 패턴과 이 것에 대한 프로젝션된 데이터베이스를 구한다는 것이다. 이 SP_ADVAN.Map 함수는 기존 SP_NAIVE.Count의 맵 함수와 동일하다. 즉 4.1절에서 설명한 맵 함수에서 고려할 사항을 똑같이 고려해 주어야 한다. 다만 Count.Map의 6행과 9행에서 SP_NAIVE.Count의 맵 함수는 값을 1로 방출했던 것과 달리 SP_ADVAN.Map 함수는 Reduce.Map의 7행에서와 같이 접미사를 값으로 방출한다. 이를 통해 SP_NAIVE.Count의 빈번한 패턴을 찾는 단계를 생략함으로써 맵리듀스 단계를 줄일 수 있다. 이를 위해 SP_NAIVE의 첫 번째 단계의 리듀스 함수에서 방출된

패턴이 빈번한지 확인하는 작업과 두 번째 단계의 리듀스 함수에서 프로젝션된 데이터베이스를 만드는 과정을 모두 SP_ADVAN.Reduce 함수에서 처리한다. SP_ADVAN.Reduce 함수에서는 일단 각 키에 대해서 방출된 값을 preValue에 저장하며 개수를 세어 본다(SP_ADVAN.Reduce의 3-4행). 이 값이 주어진 최소 지지도를 만족한다면(5행) 이 키는 빈번한 순차 패턴이므로 이 패턴과 프로젝션된 데이터베이스를 방출해야 한다. 이 프로젝션된 데이터베이스는 현재 preValue안에 저장되어 있으므로 이 값을 모두 방출하고(7행) 앞으로의 이 키에 대한 값 들도 모두 방출한다(9행). 반면 값의 개수가 최소 지지도를 만족하지 못한다면 이 접두사는 빈번한 순차 패턴이 아니라는 것을 알 수 있고 더 이상 확장 할 필요가 없으므로 이에 관련된 값들은 방출하지 않고 무시한다. 결국 이 한번의 맵리듀스 함수 호출을 통해 빈번한 패턴과 프로젝션된 데이터베이스를 구할 수 있다. 메인 함수에서는 이 결과를 getReduceOutput 함수를 통해 freqPat과 projDB에 대입하게 되고 (SP_ADVAN.Main의 5행) 다음 루프에 입력으로 사용된다.

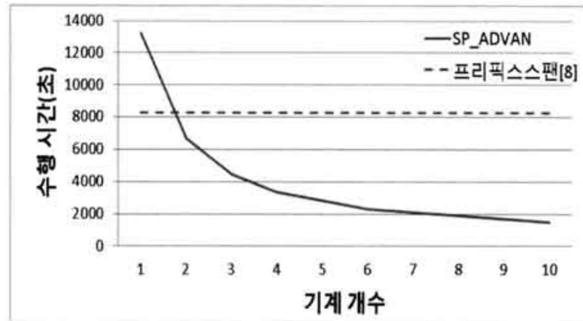
5. 실험 결과

실험 기계는 모두 10대로 Intel Core Duo 2.93Ghz 프로세서와 3GB 메모리를 탑재했다. 구현은 Java 1.6과 Hadoop 0.20을 사용했다. 데이터는 [11]의 순차 패턴 데이터 생성기를 이용한 합성 데이터를 사용하였다.

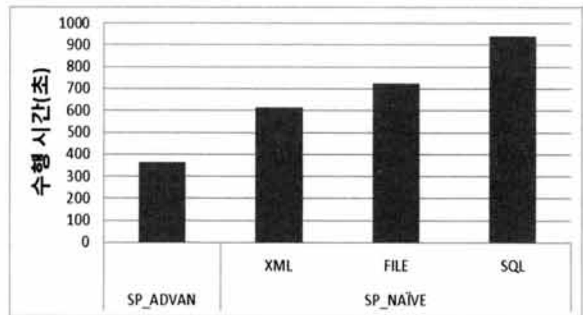
우선 제안한 방식인 SP_ADVAN의 수행 시간과 기존 프리픽스스팬[7]을 단일 기계에서 수행할 때 걸리는 시간을 비교해 보았다. 실험에 사용된 데이터는 총 10,000개의 아이템 종류가 있고 23,000,000의 시퀀스 개수를 가지며 시퀀스의 평균 길이는 10인 데이터 베이스이고 최소 지지도가 1.5%인 순차 패턴 찾는 작업을 수행했고 결과를 (그림 11)에 표시하였다. 비교를 위해 한 대의 기계에서 기존 프리픽스스팬 [7]이 수행된 시간을 점선으로 표시했다. 실험 결과 한대의 기계만 사용할 경우 기존 프리픽스스팬[7] 방식이 1.6배 정도 빠른 속도를 보이는 것을 알 수 있다. 이는 맵리듀스를 이용한 방식은 각 매퍼와 리듀서 간의 통신과 HDFS를 사용하는 추가 비용이 발생하기 때문이라 판단된다. 하지만 맵리듀스를 이용한 방식은 기계 수에 대해 선형적인 속도 증가를 보이므로 기계를 여러 대 사용할 경우 제안한 SP_ADVAN이 훨씬 좋은 성능을 보인다. (그림 11)의 실험 결과를 보면 기계를 2대만 사용해도 SP_ADVAN이 더 나은 성능을 보이는 것을 확인할 수 있다.

다음으로 SP_NAIVE와 SP_ADVAN 방식간의 수행 시간을 비교하였다. 실험 조건으로 5대의 실험 기계에서 5,000개의 아이템 종류가 있고 50,000의 시퀀스 개수를 가지는 데이터베이스에서 최소 지지도 1.2%를 만족하는 순차 패턴을 찾았다. SP_NAIVE가 SP_ADVAN보다 1.7배에서 2.6배까지 느린 것을 확인할 수 있다(그림 12).

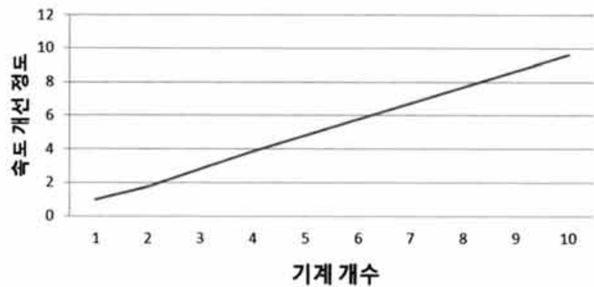
(그림 13)은 SP_ADVAN을 기계 수를 변화시켜 가며 실



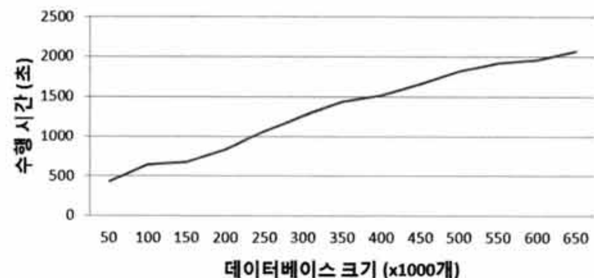
(그림 11) 프리픽스스팬[7]과 SP__ADVAN 비교



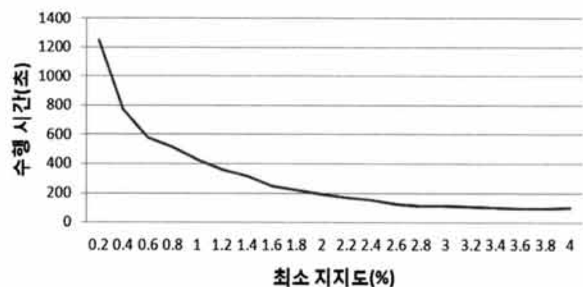
(그림 12) SP__NAIVE와 SP__ADVAN 비교



(그림 13) 기계 개수에 따른 속도 향상 비율



(그림 14) 데이터베이스의 크기에 따른 실험 결과



(그림 15) 최소 지지도에 따른 실험 결과

험해 얻은 결과의 그래프이다. 실험 데이터는 10,000개의 아이템 종류가 있고 23,000,000의 시퀀스 개수를 가지는 데이터베이스고 기계 수를 변화해 가며 최소 지지도 1.5%인 순차 패턴을 찾는 실험을 수행했다. 속도 향상 비율은 한 대의 기계로 실험하였을 때 걸리는 시간을 현재 실험의 걸리는 시간으로 나눈 값으로 실험 결과 처리해야 될 일을 각 기계가 균등하게 나누어 수행하기 때문에 예상한대로 기계 개수에 대해 거의 선형적인 성능 향상을 보였다.

(그림 14)는 SP_ADVAN에 대해 데이터베이스의 크기를 변화시켜가면서 한 실험의 결과이다. 이 실험은 5대의 실험 기계에서 5,000개의 아이템 종류가 있고 시퀀스의 평균 길이가 10인 데이터베이스에서 데이터베이스 크기를 변화시켜 가며 최소 지지도 1%인 순차 패턴을 찾은 실험이다. SP_ADVAN은 일단 프로젝션된 데이터베이스에 대해 맵 함수와 리듀스 함수를 호출하기 때문에 데이터베이스의 크기가 커지면 그만큼 방출해야 하는 용량이 커지므로 선형적으로 수행 시간이 길어지는 것을 확인할 수 있다.

마지막 (그림 15)는 SP_ADVAN에 대해 최소지지도를 변화시켜가면서 한 실험의 결과이다. 여기서는 5대의 실험 기계에서 5,000개의 아이템 종류가 있고 50,000의 시퀀스 개수를 가지며 시퀀스의 평균 길이는 10인 데이터 베이스를 사용했다. 보통 최소 지지도가 커질수록 빈번한 순차 패턴의 개수가 작아지고 길이가 짧아질 것이므로 데이터베이스를 프로젝트 하는 정도도 줄어들게 되어서 수행 시간이 적게 걸리게 되리라 예상할 수 있다. 실험 결과 최소지지도가 커질수록 더 짧은 시간이 걸림을 확인할 수 있다.

6. 결론 및 향후 연구

본 논문에서는 데이터마이닝의 중요한 분야인 순차 패턴 마이닝을 맵리듀스 프레임웍 상에서 분산 처리하는 알고리즘을 개발하였다. 한 대의 기계만을 사용하는 방식이 아닌 여러 대를 사용하는 방식을 개발하여 기계의 개수에 따라서 선형적인 속도 향상을 보이는 것을 확인할 수 있었다.

그리고 맵리듀스 프레임 상에서 각 기계들이 빈번한 순차 패턴을 공유하는 3가지 방식을 제안하고 성능을 비교하여서 하둡 상의 환경 변수(XML)로 전달하는 것이 제일 좋은 성능을 보인다는 것을 검증했다. 또 두 단계의 맵리듀스로 수행되는 방식을 한 단계로 수행되게 개선하여 최대 2.6배 정도의 빠른 속도 향상을 보였다.

향후에는 빈번한 순차 패턴을 찾는 것뿐 아니라 여러 응용의 요구에 따라 특정 제한(constraint) 조건도 만족하는 빈번한 순차 패턴을 찾는 문제를 맵리듀스 프레임웍 상에서 빠른 시간 내에 처리할 수 있는 알고리즘을 개발한다면 관련 분야에 더욱 큰 도움을 줄 수 있으리라 예상한다.

참 고 문 헌

[1] J. Dean, S. Ghemawat, "MapReduce: Simplfied Data Processing on Large Clusters," In Proc. of the 6th OSDI, 2004.

[2] Hadoop, "http://hadoop.apache.org/core/"

[3] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto. "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," In Proc. of 17th International Conference on Data Engineering, 2001.

[4] R. Agrawal, R. Srikank, "Mining Sequence Patterns," In Proc. of International Conference on Data Engineering, 1995.

[5] R. Agrawal, R. Srikant, "Fast Algorithms for Mining Association Rules," In Proc. of International Conference on Very Large Data Bases, 1994.

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

[7] J. Wang, J. Han, "BIDE: efficient mining of frequent closed sequences", In Proc. of the 20th IEEE International Conference on Data Engineering(ICDE), 2004.

[8] X. Yan, J. Han, R. Afshar, "CloSpan: mining closed sequential patterns in large datasets", In Proc. of the 3rd SIAM International Conference on Data Mining(SDB), 2004.

[9] H. Liu, J. Han, D. Xin, Z. Shao, "Mining interesting patterns from very high dimensional data: a top-down row enumeration approach", In Proc. of the 6th SIAM International Conference on Data Mining(SDM), 2006.

[10] L. Hongyan, L. Fangzhou, C. Yunjue, "New approach for the sequential pattern mining of high-dimensional sequence databases", Decision Support System, 2010.

[11] Illimine, http://illimine.cs.uiuc.edu/

김 진 현

e-mail : jhkim@kdd.snu.ac.kr

2008년 서울대학교 전기공학부(학사)

2010년 서울대학교 전기컴퓨터공학부(석사)

2010년~현재 서울대학교 전기컴퓨터공학부 박사과정

관심분야 : 데이터마이닝, 데이터베이스,

심 규 석

e-mail : shim@ee.snu.ac.kr

1986년 서울대학교 전기공학과(학사)

1988년 University of Maryland, College Park(석사)

1993년 University of Maryland, College Park(박사)

1994년~1994년 Federal Reserve Board, Research Staff

1994년~1996년 IBM Almaden Research Center, Research Staff

1996년~2000년 Bell Laboratories, Member of Technical Staff

1999년~2002년 KAIST 전산학과 조교수

2002년~현재 서울대학교 전기컴퓨터공학부 교수

관심분야 : 데이터마이닝, 데이터베이스, XML, Stream Data