비순서화된 스트림 처리를 위한 슬라이딩 윈도우 기법

(Processing Sliding Windows over Disordered Streams)

김 현 규 † 김 철 기 †† 김 명 호 †††
(Hyeon Gyu Kim) (Cheolgi Kim) (Myoung Ho Kim)

요 약 비순서화된 스트림은 슬라이딩 윈도우의 생성에 있어서 두 가지 문제점을 야기한다. 첫째는 스트림을 효율적으로 정렬하는 문제이며, 둘째는 정렬된 스트림으로부터 윈도우를 언제 생성할지 결정하는 문제이다. 본 논문에서는 이러한 문제를 해결하기 위한 윈도우 오퍼레이터의 구조와 방법에 대해 제안한다. 먼저 입력 튜플을 효율적으로 정렬하고 저장하기 위해 인덱스를 이용한 오퍼레이터의 구조를 소개한다. 그리고 윈도우의 생성 시점을 결정하기 위한 평균-기반 추정 방식을 제안한다. 제안하는 기법에서는 추정에 필요한 매개변수를 질의문에서 정의할 수 있으며, 이를 통해 사용자가 어플리케이션의 요구사항에 따라 정확성이나 응답 시간과 같은 질의 결과의 특성을 조절할 수 있도록 지원한다. 본 논문의 실험 결과는 제안한 평균-기반 방식이 기존의 연구에서 이용한 방식보다 적응성과 안정성이 우수하다는 것을 보인다.

키워드 : 스트림, 슬라이딩 윈도우, 평균-기반 추정 방식

Abstract Disordered streams cause two issues in processing sliding windows: i) how to place input tuples into a buffer in an increasing order efficiently and ii) how to determine a time point to process the windows from input tuples in the buffer. To address these issues, we propose a structure and method of operators for processing sliding windows. We first present a structure of the operators using an index to handle input tuples efficiently. Then, we propose a method to determine the time point to process the windows, which is called a *mean-based estimation*. In the proposed method, users can describe parameters required for estimation in a query specification, which provides a way for users to control the properties of query results such as the accuracy or the response time according to application requirements. Our experimental results show that the mean-based estimation provides better adaptivity and stability than the one used in the existing method.

Key words : Streams, Sliding Windows, Mean-based Estimation

1. 서 론

최근 데이타베이스 분야에서 연속 스트림을 지원하기 위한 많은 연구가 진행되었다[1-3]. 온라인 주식정보나 모니터링 정보와 같이 끊임없이 생성되어 전달되는 튜플을 실시간에 처리하는데 있어 릴레이션 개념을 적용하여 문제를 해결하고자 하였으며, 이러한 시도는 곧 DSMS(Data Stream Management System)로 구체화되었다. DSMS에서 스트림의 처리 방법은 질의언어를 이용하여 선언적으로 정의될 수 있으며, 정의된 명세로부터 생성된 질의 트리는 실행시간에 스트림 튜플의 처리에 이용된다[4,5]. DSMS의 대표적인 예로는 Stanford의 STREAM[6], Berkerly의 TelegraphCQ[7], 그리고 M.I.T와 Brown, Brandeis에서 공동으로 작업한 Aurora / Borealis 시스템[8,9] 등을 들 수 있다.

DSMS에서 슬라이딩 윈도우는 중요한 역할을 담당한다[10,11]. 예를 들어 조인(Joins)이나 집계함수(Aggregates) 등과 같은 오퍼레이터는 결과를 생성하기 위해 모든 입력 튜플을 필요로 하며, 입력 튜플이 지속적으로 전달되는 스트림의 경우에는 이용될 수 없다. 대신 슬라이딩 윈도우 오퍼레이터는 무한한 스트림에 대해 튜플의 도착시간이나 순서를 기반으로 윈도우 익스텐트(Window Extent)[10]라 불리는 유한한 부분집합으로 분리하는 역할을 담당한다. 따라서 윈도우 오퍼레이터에

† 학생회원 : 한국과학기술원 전산학과
hgkim@dbserver.kaist.ac.kr
†† 정 회 원 : 한국정보통신대학교 모바일 멀티미디어 연구소 교수
cheolgi@gmail.com
††† 종신회원 : 한국과학기술원 전산학과 교수
mhkim@dbserver.kaist.ac.kr
논문접수 : 2006년 8월 3일
심사완료 : 2006년 10월 30일

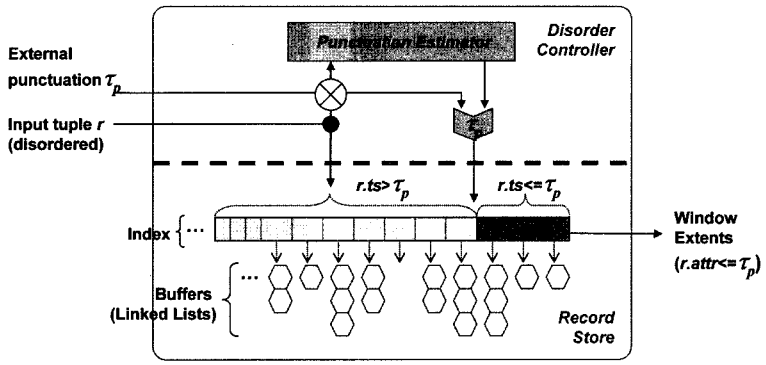


그림 1 윈도우 오퍼레이터의 구조

의해 익스텐트가 생성되면, 이를 바탕으로 조인이나 집계함수 등의 기존의 릴레이셔널 오퍼레이터를 이용하여 결과를 생성할 수 있다.

윈도우 익스텐트를 생성할 때 기존의 윈도우 오퍼레이터는 입력 튜플이 순서화되어 전달되는 것을 가정하며, 비순서화된 튜플은 무시하는 방식을 취한다[8]. 그러나 DSMS에서 튜플은 일반적으로 네트워크를 통해 전달되는 경우가 많으므로 튜플의 전달순서가 지켜지지 않을 수 있다. 비순서화된 전달은 튜플의 손실을 유발하며, 집계 질의에 있어서 결과의 질적인 저하로 이어질 수 있다. 이러한 문제를 해결하기 위해 윈도우 오퍼레이터 내부에서 버퍼를 유지하여 튜플을 정렬한 후 윈도우 익스텐트를 생성하는 작업이 필요하다.

본 논문에서는 윈도우 오퍼레이터에서 비순서화된 튜플을 효율적으로 처리하기 위한 구조와 방법을 제안한다. 윈도우 오퍼레이터에서 비순서화된 스트림을 처리하기 위해서는 두 가지 문제를 해결해야 한다. 첫째는 입력 튜플을 효율적으로 정렬하여 버퍼에 저장하는 방법이다. 윈도우 오퍼레이터는 일반적으로 질의 트리의 입력 부분에 위치하며 다량의 튜플을 처리해야 하므로, 정렬에 소요되는 노력을 줄일 필요가 있다. 둘째는 윈도우 익스텐트의 생성 시점을 결정하는 문제이다. 비순서화된 튜플을 수용하기 위해 생성 시점을 늦추는 경우 처리속도가 지연될 수 있으며, 반대의 경우는 질의 결과의 정확성에 문제가 생길 수 있다.

그림 1은 본 논문에서 제안하는 윈도우 오퍼레이터의 구조를 도식화하고 있다. 제안하는 윈도우 오퍼레이터는 레코드 저장소(Record Store)와 비순서 제어기(Disorder Controller)로 구성된다. 레코드 저장소는 버퍼와 인덱스로 구성되어 있으며, 인덱스는 전달된 튜플을 순서화하여 버퍼에 저장하는 역할을 담당한다. 그리고 비순서 제어기는 윈도우 익스텐트의 생성 시점을 추정하기 위한 모듈이며, 전달된 튜플의 정보를 바탕으로 구두점(Punctuations)[12][1]을 생성한다. 구두점의 생성을 위해 본 논문에서는 평균-기반 추정방식(Mean-based estimation)을 제안한다. 평균-기반 방식에서는 추정에 필요한 매개변수를 질의문에서 정의할 수 있으며, 이를 통해 사용자가 어플리케이션의 요구사항에 따라 정확성이나 응답 시간과 같은 질의 결과의 특성을 조절할 수 있도록 지원한다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 슬라이딩 윈도우 처리 방식과 관련한 기존 연구를 소개한다. 3장에서는 비순서화된 스트림으로부터 윈도우를 생성하기 위한 질의 구문을 제안한다. 그리고 4장에서는 레코드 저장소에서 인덱스를 이용한 튜플의 저장과 처리 방법을 제안하며, 5장에서는 비순서 제어기에서 평균-기반 방식을 기반으로 윈도우 생성 시점을 추정하기 위한 방법에 대해 설명한다. 6장에서는 실험을 통해 평균-기반 방식과 기존의 추정 방식을 비교한다. 마지막으로 7장에서는 추후연구 방향을 제시하고 결론으로 마무리한다.

2. 관련 연구

비순서화된 스트림을 처리하기 위해 기존에는 크게 두 가지 접근 방법이 이용되었다[13]. 첫째는 튜플을 정렬하기 위해 버퍼를 이용하는 방법이며, 둘째는 구두점을 이용하는 방법이다. 전자는 Aurora[8]에서 이용되었으며, 후자는 STREAM[6]이나 NiagaraCQ[14] 등에서 이용되었다.

Aurora[8]에서 윈도우 오퍼레이터는 비순서화된 튜플을 단순히 무시하는 방법을 취한다. 비순서화된 튜플의 손실을 막기 위해 윈도우 오퍼레이터 앞에 슬랙(Slack)

1) 구두점(Punctuations)이란 어느 시점에 시스템에 구두점 τ_p가 전달되었을 때, 해당 시점 이후로는 τ_p보다 적은 값을 지니는 튜플이 전달되지 않음을 의미한다. 따라서 구두점 τ_p가 전달되거나 식별되었을 때 윈도우 오퍼레이터에서는 τ_p보다 적은 값을 지니는 튜플을 모두 처리할 수 있다.

이라 불리는 버퍼를 위치시켜 튜플을 정렬할 수 있도록 지원한다. Aurora는 네트워크 지연(Latency)의 바운드가 미리 알려져 있다는 가정하에 해당 바운드를 커버할 수 있는 충분히 큰 고정 크기의 슬랙 버퍼를 이용하도록 하였다. 네트워크 지연의 바운드가 미리 알려져 있지 않을 경우 고정 크기의 버퍼를 이용하는 것은 한계가 있을 수 있다.

STREAM[6]에서 윈도우 오퍼레이터는 비순서 처리를 위한 메커니즘을 가지지 않는다. 대신 네트워크로부터 전달되는 튜플은 입력 관리자(Input Manager)[15,16]로 불리는 외부 모듈에서 정렬된 후 DSMS로 전달된다. 입력 관리자는 전달되는 튜플을 정렬하여 버퍼에 저장하며, 저장된 튜플로부터 윈도우를 생성하기 위한 시점을 결정하기 위해 현재까지 발생한 네트워크 지연의 최대값에 근거하여 구두점을 추정한다. 그러나 해당 추정 방식은 시간을 기반으로 정의된 슬라이딩 윈도우에 한해 이용될 수 있다.

NiagaraCQ[14]는 스트림 소스로부터 전달되는 구두점을 이용하거나, 네트워크 지연의 최대값을 미리 안다는 가정으로부터 구두점을 추정하는 방식을 이용한다. Gigascope[17] 역시 네트워크 지연의 최대값이 일정하다는 가정(Banded-increasing property)으로부터 일정 시간 동안 데이타의 처리를 지연시키는 방법을 이용한다.

WID 방식[10]에서는 윈도우를 기반으로 한 집계 질의를 효율적으로 처리하기 위한 기법에 대해 소개하였으며, 비순서화된 튜플을 처리하기 위해 스트림 소스에서 전달되는 구두점 정보를 이용하였다. 그러나 스트림 소스로부터 네트워크를 통해 전달되는 구두점은 네트워크 지연에 의해 역시 비순서화될 수 있으므로 한계가 있을 수 있다. 그들의 연구에서 구두점을 내부적으로 추정하는 방식은 언급하지 않고 있다.

3. 윈도우 명세

슬라이딩 윈도우 오퍼레이터는 무한한 스트림을 윈도우 익스텐트라 불리는 유한한 서브셋으로 분리하는 역할을 수행한다. 예를 들어 환경 센서로부터 전달되는 기온 값에서 최근 5분 동안의 최대값을 구하기 위한 질의를 가정해보자. 최대값은 1분마다 갱신된다고 가정한다. 이와 같은 질의는 슬라이딩 윈도우와 함께 Q1으로 정의될 수 있다. Q1에서는 CQL[18]과 유사한 질의 구문을 이용하였으며, 윈도우의 정의를 위해 WID 방식[10]에서 제안한 구문을 이용하였다. WID 방식에서 윈도우의 정의 구문은 RANGE, SLIDE, WATTR 등으로 구성되며, 각각 윈도우 익스텐트의 크기 및 해당 익스텐트의 슬라이드 주기, 그리고 익스텐트의 크기와 슬라이드가 어떤 애트리뷰트의 값을 기준으로 설정될지를 나타낸다.

```
Q1: SELECT max(temp)
    FROM Sensors [RANGE 5 minutes
                  SLIDE 1 minute
                  WATTR ts]
```

위 질의에서 센서 값의 스키마는 <*ts*, *temp*>라고 가정하였으며, 각각 센싱된 시점의 타임스탬프와 온도값을 의미한다. 튜플의 타임스탬프는 두 가지 방식으로 할당될 수 있다[15]. 첫째는 시스템에 전달될 때 시스템에 의해 할당되는 타임스탬프이며, 둘째는 튜플이 생성될 때 센서와 같은 스트림 소스에 의해 할당되는 타임스탬프에 해당한다. 편의상 전자를 시스템 타임스탬프(System timestamp) 그리고 후자를 어플리케이션 타임스탬프(Application timestamp)라 한다. Q1에서 *ts*는 어플리케이션 타임스탬프에 해당한다.

버퍼를 이용하여 비순서화된 스트림을 처리할 때, 일반적으로 주어진 버퍼의 크기로 얼마만큼의 튜플 손실을 막을 수 있을지 알아내기는 힘들다. 예를 들어 위 질의와 스키마를 바탕으로 {<2, 10>, <1, 20>, <4, 10>, <3, 25>, <5, 15>}와 같은 입력 스트림의 시퀀스를 가정해보자. (시퀀스에서 가장 왼쪽의 튜플이 가장 먼저 전달되었다고 가정하며, 오른쪽으로 갈수록 후에 전달된 것으로 간주한다.) 해당 시퀀스(보다 간소화된 표현으로) {2, 1, 4, 3, 5}에서는 크기 2인 슬랙 버퍼를 이용하여 그림 2(a)와 같이 비순서화된 모든 튜플을 구할 수 있다. 한 가지 주의할 점은 동일한 버퍼로 다른 형태의 시퀀스에서 비순서화된 튜플을 모두 구한다는 보장은 없다는 사실이다. 그림 2(b)에서 볼 수 있듯이, 동일한 버퍼를 이용했을 때 시퀀스{2, 1, 4, 5, 3}에서 5번째 튜플은 버려지게 되며, 전체 데이타에 대한 튜플의 손실율(Drop ratio)은 20%(=1/5)가 된다. 마찬가지로 동일한 버퍼에서 시퀀스{2, 4, 1, 5, 3}은 40%(=2/5)의 손실율을 보이며(그림 2(c)), 시퀀스{4, 5, 1, 2, 3}은 60%(=3/5)의 손실율을 보인다(그림 2(d)).

본 논문에서 제안하는 방식에서는 비순서화된 튜플을 구하기 위해, 질의문에서 버퍼의 크기를 기술하기 보다는 사용자가 원하는 튜플의 손실율(Drop ratio)을 정의할 수 있도록 지원한다. 손실율이 주어지면 버퍼의 크기는 주어진 손실율을 만족시키기 위해 가변적으로 변하게 된다. 제안하는 방식에서는 튜플의 손실율을 질의문에서 기술하기 위한 부가적인(Optional) 파라미터로서 *DRATIO*를 정의하였다. 사용자는 DRATIO의 정의를 통해 어플리케이션의 특성에 따라 튜플의 손실을 조절할 수 있다. 예를 들어 정확성이 요구되는 어플리케이션의 경우 DRATIO의 값을 적게 주거나, 반대로 빠른 처리속도가 요구되는 어플리케이션의 경우 DRATIO의 값

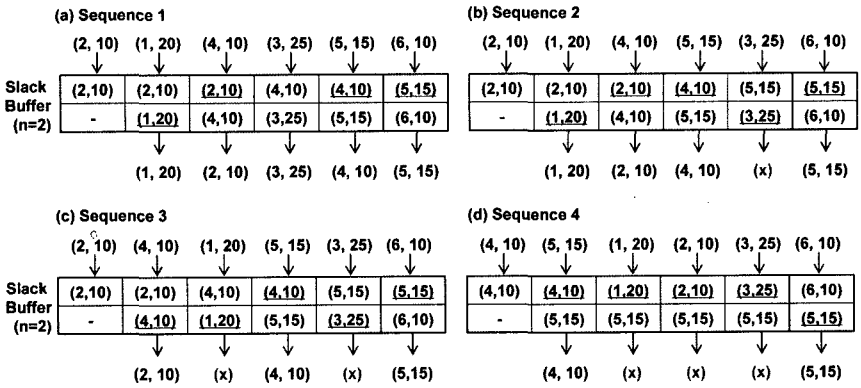


그림 2 입력 시퀀스에 따른 튜플의 손실율

을 크게 줄 수 있다. Q2는 DRATIO를 정의한 예를 보여준다. DRATIO의 값은 퍼센티지 단위로 정의된다.

```
Q2: SELECT max(temp)
    FROM Sensors [RANGE 5 minutes
                  SLIDE 1 minute
                  WATTR ts]
                  DRATIO 1%]
```

제안하는 방식에서는 DRATIO 이외에 고정 크기의 슬랙 버퍼를 이용하여 비순서화된 튜플을 정렬할 수 있도록 지원한다. 이를 위해 *SLACK*이라는 부가적인 파라미터를 정의하였으며, 네트워크 지연의 바운드가 미리 알려진 경우 효율적으로 이용될 수 있다. Q3는 SLACK을 이용한 예를 보여준다. SLACK의 값은 버퍼의 크기를 나타낸다. SLACK과 DRATIO는 상호 배제적으로 정의된다.

```
Q3: SELECT max(temp)
    FROM Sensors [RANGE 5 minutes
                  SLIDE 1 minute
                  WATTR ts]
                  SLACK 10]
```

비순서를 제어하기 위해 정의되는 파라미터 외에, 질의문에서 튜플의 저장에 이용되는 버퍼 크기의 최대값을 제한할 수 있다. BSIZE는 레코드 저장소에서 허용가능한 버퍼 크기의 최대값을 의미하며, 아래와 같이 SLACK이나 DRATIO와 함께 이용될 수 있다.

```
Q4: SELECT max(temp)
    FROM Sensors [RANGE 5 minutes
                  SLIDE 1 minute
                  WATTR ts]
                  DRATIO 1%
                  BSIZE 100]
```

만약 BSIZE가 DRATIO나 SLACK과 함께 정의되는 경우, 우선 순위는 BSIZE에 있다. 예를 들어 Q4가 주어졌을 때 실시간에 비순서 제어기는 DRATIO 1%를 기준으로 추정을 수행하게 되며, 추정 과정에서 버퍼의 크기가 100을 넘어서게 되면 버퍼는 100으로 고정된다. 이러한 경우 주어진 DRATIO는 버퍼의 크기가 다시 100 이하로 돌아오기 전까지는 동작하지 않는다.

4. 윈도우 처리

슬라이딩 윈도우 오퍼레이터는 일반적으로 질의 트리의 입력 부분에 위치하며, 다량의 입력 튜플을 실시간에 처리하도록 요구된다. 따라서 전달된 튜플을 순서화하고 저장하는데 들어가는 비용을 최소화할 필요가 있다. 예를 들어 그림 2에서 보인 바와 같이 입력 튜플을 순서화하기 위해 정렬을 이용할 경우, 버퍼의 크기가 커진다면 튜플이 전달될 때마다 정렬하는 것은 시스템에 과부하를 줄 수 있다.

이러한 문제를 해결하기 위해 본 논문에서는 인덱스와 윈도우 패인(Window Pane)[11] 개념을 이용한 윈도우 오퍼레이터의 구조를 제안한다. 인덱스는 순환 배열(Circular array)의 형태를 지니며, 각 배열의 인자는 하나의 윈도우 패인을 가리키는 포인터로 구성된다. 윈도우 패인은 그림 3과 같이 윈도우의 분리된 세그먼트(Disjointed segments)에 해당하며, 그림 3에서는 4개의 윈도우 패인으로 구성된 윈도우 익스텐트를 보여준다. 제안하는 구조에서 윈도우 패인은 입력 튜플을 저장하는 백(Bag)에 해당한다. 윈도우 패인의 크기는 GCD (*rangeVal*, *slideVal*)의 형태로 계산될 수 있으며, rangeVal과 slideVal은 윈도우 명세에서 각각 RANGE와 SLIDE의 값에 해당한다. 예를 들어 Q1~Q4의 경우

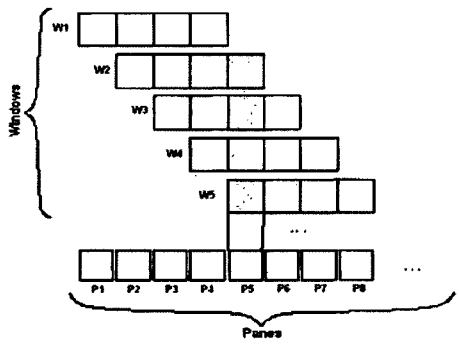


그림 3 4개의 패인(Pane)으로 구성된 윈도우 익스텐트

하나의 윈도우 패인의 크기는 60초가 된다. 즉 1분 내에 들어오는 모든 튜플은 같은 패인에 저장된다. 일반적으로, 타임스탬프를 기준 어트리뷰트로 하는 윈도우에서 윈도우 패인의 최소 크기는 1초가 된다. 이는 대부분의 슬라이딩 윈도우 명세가 초 단위로 RANGE나 SLIDE를 기술하도록 지원한다는 사실로부터 기인한다.

튜플이 전달될 때마다 레코드 저장소 내의 변환기(Mapper)는 전달된 튜플이 어느 윈도우 패인에 저장되어야 하는지 결정하는 역할을 담당한다. 이러한 결정은 전달된 튜플의 타임스탬프를 윈도우 패인의 크기로 나눔으로써 해결될 수 있다. 그림 4는 전달된 튜플이 윈도우 패인에 저장되는 프로세스를 보여준다. 한 가지 주목할 점은 모든 튜플의 순서를 정확히 지킬 필요는 없다는 점이다. 윈도우 오퍼레이터에서 튜플의 순서를 유지하는 목적은 윈도우 익스텐트를 명확히 구분하기 위한 것이며, 패인 단위의 순서만 명확히 지켜질 경우 익스텐트를 구분할 수 있다. 따라서 제안하는 구조에서 윈도우 패인 내의 튜플의 순서는 보장되지 않는다.

윈도우 오퍼레이터는 현재 유지되는 윈도우 패인으로부터 익스텐트의 생성 시점을 결정하기 위해 비순서 제어기에서 추정된 구두점을 이용한다. 우선 윈도우 익스텐트의 범위는 질의 명세로부터 RANGE가 n이고 SLIDE가 s로 주어졌을 때 식 (1)과 같이 기술될 수 있다. 그리고 식 (2)는 비순서 제어기로부터 구두점 τ_p가 전달되었을 때 익스텐트의 생성 시점을 결정하기 위한 조건이다. 이는 윈도우 익스텐트의 가장 큰 타임스탬프가 구두점 τ_p보다 적거나 같은 경우를 의미한다. 식 (1)과 식 (2)에서 R은 튜플의 백(Bag)을 의미하며, r은 R 내의 튜플을 나타낸다. 그리고 $wattr$는 r의 기준 어트리뷰트를 의미한다.

$$\textbf{\textit{extent(i)}} = \{r \in \mathrm{R} \mid (i+1)*s - n \le r.wattr < (i+1)*s\} \quad (1)$$

$$(i+1)*s \le \tau_p \quad (2)$$

식 (2)의 조건을 바탕으로 익스텐트를 일관성 있게 생성하기 위해서는 전달되는 구두점의 값이 항상 오름차순(Monotonously increasing fashion)을 유지해야 한다. 이것이 지켜지지 않을 경우 하나의 윈도우 패인이 여러 개의 익스텐트에 포함될 수 있다. 이는 새로이 추정된 구두점이 이전 값보다 적은 경우 비순서 제어기에서는 이전 값을 보내도록 설계함으로써 해결될 수 있다.

윈도우 익스텐트를 생성할 때, 윈도우 오퍼레이터는 익스텐트의 끝을 나타내는 경계 튜플(Boundary tuple)을 추가하여, 조인이나 집계함수 등의 다른 오퍼레이터에서 익스텐트의 범위를 판단할 수 있도록 지원한다. 경계 튜플은 범위 판단을 위한 목적 외에 조인 오퍼레이터에서 입력 스트림간의 동기화를 위해 이용될 수도 있다. 예를 들어 도로의 상황에 따라 도로를 이용하는 차량에 대해 과징금을 부과할지 아닐지 판단하는 질의를 가정해보자. 도로에서 최근 5분의 평균속도가 40 mph 이하이면 혼잡상황으로 간주하고, 도로내의 차량에 대해 일정량의 과징금을 부과한다고 하자. 그리고 각 차량은 매 30초마다 센서를 통해 자신의 속도를 알려준다고 가정한다. 그림 5는 이러한 질의에 대한 트리를 간소화한

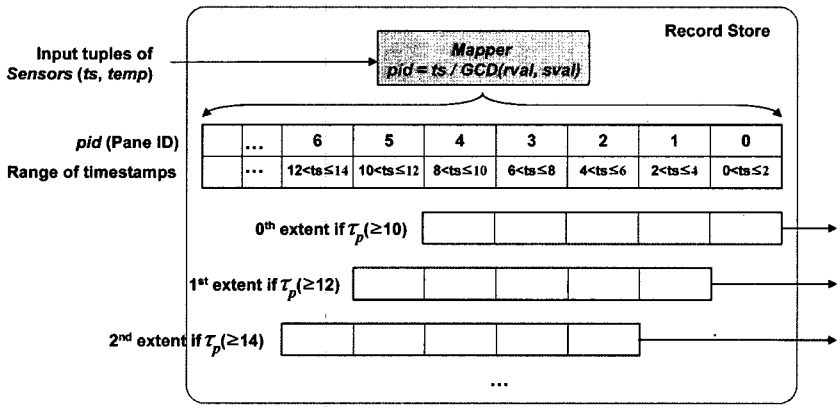


그림 4 튜플의 저장과 윈도우 익스텐트의 생성

형태로 보여준다. 그림 5에서 타원은 오퍼레이터를 의미하며, OP_SRC는 입력 튜플을 받아들이기 위한 오퍼레이터, OP_WIN은 윈도우 오퍼레이터, OP_AVG는 평균을 구하기 위한 집계 오퍼레이터, 그리고 OP_JOIN은 조인 오퍼레이터를 나타낸다.

그림 5에서 Path I는 입력 튜플로부터 최근 30초 내의 데이타를 윈도우로 유지하여, 현재 도로에 있는 차량 정보를 구하는 부분에 해당한다. 그리고 Path II는 현재 도로가 혼잡 상황인지 아닌지 체크하기 위한 부분이다. Path I의 결과와 Path II의 결과를 조인하면 각 차량에 대해 과징금의 부과 여부를 판단할 수 있다. 그러나 만약 두 스트림이 동기화되어 있지 않다면 부정확한 질의 결과가 발생할 수 있다. 예를 들어 어느 과거 시점에 Path II의 결과에서 혼잡 상황으로 판단된 이후 도로의 상황이 좋아지고 있다고 가정해보자. 이러한 상황에서 Path I만 지속적으로 수행된다면, 실제로는 도로의 상황이 좋은데도 불구하고 과징금을 부과하는 결과가 생길 수 있다. 경계 튜플은 동기화를 가능하게 하여, 부정확한 질의 결과를 방지하는데 도움을 줄 수 있다.

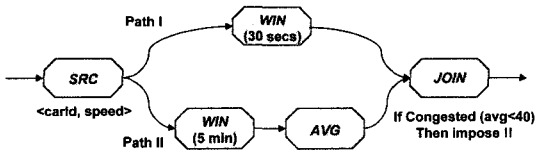


그림 5 도로 상황에 따라 과징금을 부과하기 위한 질의 트리

5. 평균-기반 추정 방법

비순서 제어기는 레코드 저장소 내의 윈도우 패인으로부터 익스텐트를 생성하기 위한 시점을 결정하기 위해 구두점을 생성한다. 구두점의 추정에 있어서 기존에는 네트워크에서 발생한 지연(Latency)의 최대값을 반영하는 방법[11,15]을 이용하였다. 이러한 방식에서는 튜플이 전달될 때마다 시스템에 도착한 시간과 튜플이 생성된 시간의 차이, 즉 튜플의 시스템 타임스탬프와 어플리케이션 타임스탬프를 계산하며, 계산된 차이값 중 최대값을 항상 유지한다. 그리고 튜플이 전달될 때마다 튜플의 시스템 타임스탬프에서 계산된 최대값을 뺀 값을 구하여 구두점으로 활용한다. 이러한 방식은 현재까지 발생한 네트워크 지연의 최대값보다 더 큰 비순서(Disorder)가 발생하지 않을 것임을 가정하고 구두점을 추정하는 방식에 해당한다. 편의상 이와 같은 방식을 지연값-기반 추정 방식(Latency-based estimation)으로 명명한다.

그림 6은 스트림 시퀀스가 {<2, 10>, <1, 20>, <4, 10>, <3, 25>, <5, 15>, <6, 10>, <7, 10>}으로 주어졌을 때 Q1에 대해 지연값-기반 방식의 동작 상황을 도식화하고 있다. 그림 6에서 *maxDelay*는 네트워크 지연의 최대값을 의미하며, 주어진 시퀀스 이전의 스트림에서 발생한 지연의 최대값을 3으로 가정하였다. maxDelay 값은 5번째 튜플 <3, 25>가 들어올 때 4로 바뀐다. 이는 5번째 튜플의 시스템 타임스탬프가 7이고 어플리케이션 타임스탬프는 3이므로 지연 값이 4가 되며, 기존의 최대값 3보다 크기 때문이다. 구두점은 단순히 전달된 튜플의 시스템 타임스탬프와 지연의 최대값의 차이를 통해 구할 수 있다. 그리고 7번째 튜플이 전달되었을 때, 조건 식 (2)를 만족하므로 윈도우 익스텐트가 생성될 수 있다.

위에서 설명한 내용에 대해 식으로 정리하면 식 (3)과 같다. 아래에서 r_i는 i번째 튜플을 의미하며, n은 가장 최근에 전달된 튜플의 순서를 의미한다. 그리고 $sysTS$는 시스템 타임스탬프를, 그리고 $appTS$는 어플리케이션 타임스탬프를 나타낸다. τ_p는 구두점을 의미한다.

$$maxDelay = \max(r_i.sysTS - r_i.appTS) \quad (\text{where } 0 \le i \le n) \qquad (3)$$

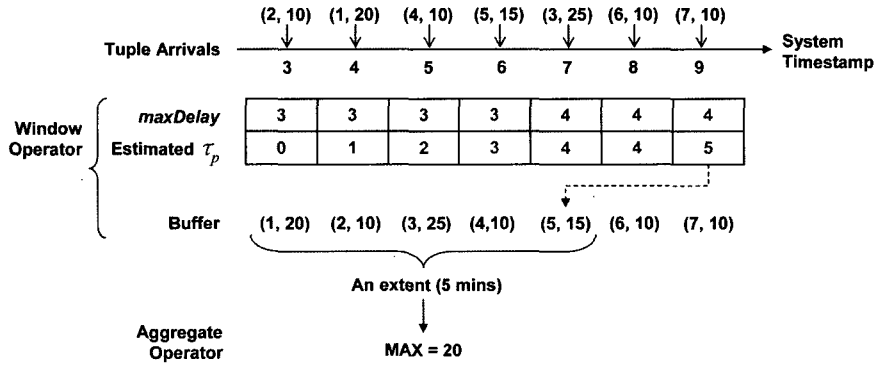


그림 6 지연값-기반 추정 방식의 동작 상황

$$\tau_p = r_n.sysTS - maxDelay \quad (4)$$

지연값-기반 방식은 응용성(Applicability)과 안정성(Stability)의 두 가지 관점에서 문제가 있을 수 있다. 먼저 응용성의 관점에서 해당 방법은 윈도우의 기준 애트리뷰트가 시간인 윈도우에만 국한되어 이용될 수 있다. 다음으로 안정성의 관점에서 만약 초기에 예외적인 최대값이 발견될 경우, 이후의 모든 τ_p 는 해당 최대값을 적용 받아 버퍼사이즈와 튜플의 대기시간이 증가할 수 있다.

이와 같은 문제점을 보완하기 위해 본 논문에서는 평균-기반 추정 방식(Mean-based estimation)을 제안한다. 평균-기반 추정 방식은 지연값-기반 방식과 유사하며, 비순서의 최대값을 구하기 위해 튜플의 타임스탬프 값을 이용하는 대신 튜플의 기준 어트리뷰트 값의 평균을 이용한다. 평균-기반 추정 방식을 식으로 나타내면 식 (5)에서 식 (6)과 같다. 아래에서 $wattr$는 튜플 r_i의 기준 어트리뷰트를 나타내며, μ 는 최근 전달된 30개의 튜플들의 기준 어트리뷰트 값의 평균을 의미한다. 그리고 $maxDisorder$는 비순서의 최대값을 나타낸다.

$$maxDisorder = \max(\mu - r_i.wattr) \quad (\text{where } 0 \le i \le n) \quad (5)$$

$$\tau_p = \mu - maxDisorder \quad (6)$$

$$where \ \mu = \frac{\sum_{j=n-qsize}^{n} r_j.wattr}{qsize} \ (where \ qsize = 30)$$

평균-기반 추정 방식은 기준 어트리뷰트의 평균을 이용하기 때문에 윈도우의 타입에 상관없이 범용적으로 이용될 수 있다. 그러나 위 식은 아직 안정성의 관점에서는 문제가 있을 수 있다. 역시 초기에 예외적인 최대값이 발생할 경우 이후의 모든 τ_p는 예외적인 최대값을 적용 받아 버퍼사이즈와 튜플의 대기시간이 증가할 수 있다. 이러한 문제는 최근 계산된 $maxDisorder$들의 평균을 이용함으로써 해결될 수 있다. 따라서 식 (6)은 식 (7)과 같이 재정의된다.

$$\tau_p = \mu - maxDisorderAvg \quad (7)$$

$$where \ \mu = \frac{\sum_{k=n-dsize}^{n} maxDisorder_k}{dsize} \ (where \ dsize = 10)$$

이 외에 평균-기반 방식은 질의문에서 주어진 손실율(Drop ratio)을 만족할 수 있도록 τ_p를 추정해야 한다. 이와 관련하여 본 논문에서 제안하는 방식은 휴리스틱을 이용하는 방식이다. 예를 들어 추정된 τ_p를 이용하였을 때 실제 손실율이 주어진 손실율보다 적을 경우는 추정된 값보다 큰 τ_p를 윈도우 생성에 적용할 수 있다. 반대로 실제 손실율이 더욱 커지는 경우는 τ_p의 값을 줄여야 한다. 식 (8)은 이를 반영한 식에 해당한다. 아래에서 ofs은 손실율 비교 결과에 따라 추정된 τ_p의 값을 조정하기 위한 변수에 해당한다. ofs의 조정은 100개의 튜플이 전달될 때마다 이루어진다. 만약 전달된 100개의 튜플에서 일어난 손실 개수가 질의문에서 주어진 손실율보다 적으면 ofs을 증가시킨다. 반대의 경우는 ofs을 감소시킨다.

$$\tau_p = \mu - maxDisorderAvg + ofs \quad (8)$$

윈도우의 기준 어트리뷰트가 시간일 경우, 주어진 손실율을 만족하기 위한 식은 확률론적인 접근을 통해 좀 더 체계적으로 구해질 수 있으며, [19]에서 언급된 바 있다.

6. 실험 결과

이 장에서는 실험을 통해 적응성과 안정성 측면에서 평균-기반(Mean-based) 방식과 지연값-기반(Latency-based) 방식을 비교하였다. 첫번째 실험은 평균-기반 방식이 주어진 손실율을 만족하는지 확인하며, 두번째 실험은 버퍼의 크기나 튜플의 대기시간 관점에서 두 방식을 비교하였다. 마지막 실험은 예외적인 크기의 비순서가 발생할 경우 두 방식의 추정 결과가 어떤 영향을 받는지 비교하였다.

실험을 위해 데이타 생성기(Data Generator)를 직접 구현하였다. 구현된 데이타 생성기는 스트림 소스에서 데이타의 생성이 임의로 이루어지며, 네트워크의 지연은 주어진 바운드 내에서 정규 분포를 따르도록 구성하였다. 현재 많이 이용되고 있는 TinyDB[20]나 SENSIM[21] 등을 이용해 데이타를 생성해 보았으나, 실험에 적용하기에는 한계가 있었다. 예를 들어 TinyDB[20]의 경우 네트워크 지연의 바운드가 꾸준히 증가하는 경향을 보였으며, 네트워크 지연의 바운드에 따른 실험 결과를 비교하기가 어려웠다. 그리고 SENSIM[21]이나 NS2 센서 네트워크 확장 버전[22]은 비순서화된 튜플이 발생하지 않았다. 이는 두 시뮬레이터에서 MAC 계층으로 802.11을 이용한 것에 기인하였으며, 전송 속도가 센서의 액티베이션 속도보다 빨랐다. 본 실험은 리눅스 7.3을 탑재한 인텔 펜티엄 4.2.4 MHz 머신에서 수행되었다. 실험에 이용된 각 데이타 세트는 6M 정도의 크기를 지니며, 약 500,000개 정도의 튜플로 구성되었다.

첫번째 실험은 평균-기반 방식이 주어진 손실율을 만족하는지 확인하고자 하였다. 실험을 위해 질의문에서 DRATIO를 1%에서 15%까지 주고 실험을 수행하였으며, 그림 7과 같은 실험결과를 얻었다. 그림 7의 표에서 왼쪽 열은 비순서의 바운드에 해당하며 초 단위로 표현되었다. 그리고 표의 가장 아래 행은 각 열의 평균 결과를 나타낸다. 실험 결과에서 주어진 DRATIO를 위반하는 실험결과는 발생하지 않았다. 그러나 5% 보다 큰

DRATIO에 대해 손실율 결과가 주어진 손실율에 비해 다소 차이가 생긴다는 사실을 알 수 있었다. 이와 같은 적응성 문제는 추후에 이론적인 접근을 통해 해결되어야 할 것으로 예상된다.

다음 실험은 버퍼의 크기나 튜플의 대기시간 관점에서 두 방식을 비교하였다. 튜플의 대기시간은 오퍼레이터 내의 버퍼에서 각 튜플의 평균 대기시간을 의미한다. 실험을 위해 비순서의 바운드를 서서히 증가시켰을 때, 두 방식에서 버퍼 사이즈나 평균 대기시간이 어떻게 변화하는지 확인하고자 하였다. 그림 8은 이에 대한 실험 결과를 보여준다.

그림 8(a)는 두 방식에서 추정에 이용된 버퍼의 크기를 보여준다. 결과에서 볼 수 있듯이 바운드가 7초보다 적은 경우 사용된 버퍼의 크기는 비슷하나, 8초를 넘어서면서부터 차이가 커지기 시작하였다. 예를 들어 바운드가 10초일 경우, 차이가 900개까지 커진다. 그림 8(b)는 두 방식에서 평균 대기시간의 변화를 보여준다. 지연값-기반 방식의 경우 대기시간이 꾸준히 증가하는 반면, 평균-기반 방식의 경우 대기시간의 증가폭이 크지 않음을 알 수 있다. 이와 같은 실험 내용으로부터 평균-기반 방식이 지연값-기반 방식에 비해 상대적으로 나은 적응성을 가짐을 알 수 있다.

Maximum bound of the disorder (secs)	DRATIOs (%)				
	15	10	5	2.5	1
...			...		
5	6.51	5.32	2.55	1.61	0.59
6	9.21	6.22	3.28	1.84	0.61
7	9.91	6.39	3.30	2.22	0.71
8	8.87	6.82	3.38	1.96	0.74
9	9.87	7.37	3.52	2.12	0.73
...			...		
Average (%)	8.87	6.42	3.21	1.95	0.68

그림 7 평균-기반 방식의 실험 결과

마지막으로 안정성의 관점에서 실험을 위해 임의로 데이타에 예외적인 크기의 비순서를 위치시켜 보았다. 그리고 비순서의 크기를 증가시킬 때 버퍼의 크기나 평균 대기시간에 어떠한 결과를 미치는지 알아보고자 하였다. 그림 9는 임의의 데이타 세트에서 초반에 하나의 튜플에 대해 비순서의 크기를 증가시켜본 결과를 보여주고 있다. 그림 7에서 X축은 비순서의 바운드를 초단위로 나타내며, Y축은 버퍼의 크기 및 평균 대기시간을 나타낸다. 실험 결과로서 지연값-기반 방식의 경우 주어진 비순서 크기에 민감한 반응을 보이며, 평균-기반 방식에서는 이와는 반대로 거의 영향을 받지 않음을 알 수 있었다.

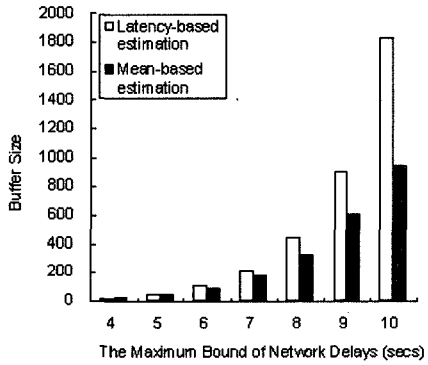


그림 8(a) 버퍼 크기의 비교

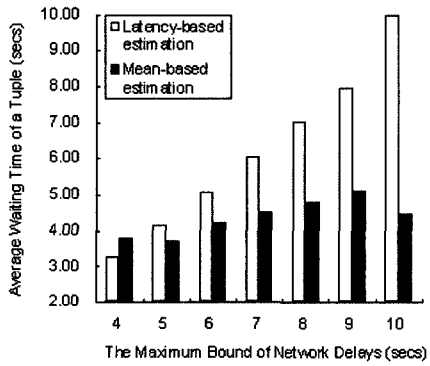
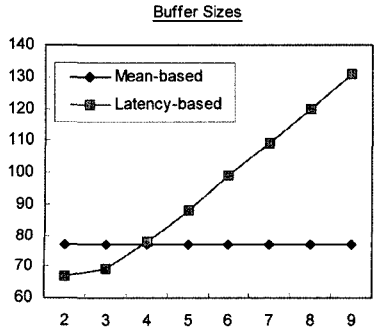


그림 8(b) 평균 대기시간의 비교



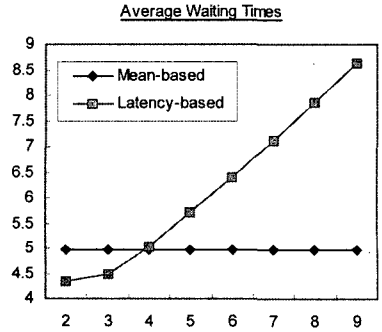


그림 9 비순서 크기에 따른 버퍼 크기 및 평균 대기시간의 증가

7. 결 론

본 논문에서는 슬라이딩 윈도우를 효율적으로 처리하기 위한 구조와 방식에 대해 소개하였다. 본 논문에서 제시한 슬라이딩 윈도우 오퍼레이터는 레코드 저장소와 비순서 제어기의 두 가지 모듈로 구성되어 있다. 레코드 저장소에서는 전달된 튜플을 효율적으로 순서화하기 위해 인덱스와 윈도우 패인을 이용하였다. 그리고 비순서 제어기에서는 윈도우 익스텐트의 생성 시점을 판단하는 역할을 하며, 평균-기반 추정 방식을 이용하였다. 평균-기반 방식에서는 추정에 필요한 파라미터인 DRATIO를 질의문에서 정의할 수 있도록 지원하며, 이를 통해 사용자는 질의문에서 DRATIO를 기술함으로써 어플리케이션의 요구에 따라 정확성이나 응답시간 등의 질의 결과의 특성을 조정할 수 있다. 추정에 이용된 평균-기반 방식에 대해서는 실험을 통해 적응성과 안정성의 관점에서 기존의 지연값-기반 방식에 비해 우수함을 보이고자 하였다. 추후에는 비순서 제어를 위한 추정식에 있어서 이론적인 접근을 통해 적응성을 높이는 방향으로 연구를 진행할 예정이다.

참 고 문 헌

[1] Douglas Terry, David Goldberg, David Nichols, and Brian Oki, *Continuous Queries over Append-Only Databases*. ACM SIGMOD, 1992.

[2] Samuel R. Madden, Mehul A. Shah, Joseph M. Hellerstein and Vijayshankar Raman, *Continuously Adaptive Continuous Queries over Streams*. ACM SIGMOD Conference, Madison, WI, June 2002.

[3] S. Babu and J. Widom, *Continuous Queries over Data Streams*. ACM SIGMOD Record, Sep. 2001.

[4] Rajeev Motwani et al, *Query Proessing, Resource Management, and Approximation in a Data Stream Management System*. CIDR 2003, Jan. 2003.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, *Models and Issues in Data Stream Systems*. Invited paper in Proc. of the 2002 ACM Symp. on Principles of Database Systems (PODS 2002), June 2002.

[6] Arvind Arasu et al, *STREAM: The Stanford Data Stream Management System*. IEEE Data Engineering Bulletin, Vol. 26 No. 1, March 2003.

[7] Sirish Chandrasekaran et al, *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World*. CIDR 2003.

[8] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. *Aurora: A New Model and Architecture for Data Stream Management*. VLDB Journal (12)2: 120-139, August 2003.

[9] D. Abadi at al, *The Design of the Borealis Stream Processing Engine*. CIDR 2005, Asilomar, CA, January 2005.

[10] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker, *Semantics and Evaluation Techniques for Window Aggregates in Data Streams*. ACM SIGMOD 2005, June 14-16, 2005, Baltimore, Maryland, USA.

[11] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker, *No Pane, No Gain: Efficient Evaluation of Sliding·Window Aggregates over Data Streams*. SIGMOD Record, Vol 34, No. 1, March 2005.

[12] Peter A. Tucker, David Maier, Time Sheard, Leonidas Fegaras, *Exploiting Punctuation Semantics in Continuous Data Streams*. IEEE Transactions on Knowledge and Data Engineering, May/June 2003.

[13] David Maier, Jin Li, Peter A. Tucker, Kristin Tufte and Vassilis Papadimos, *Semantics of Data Streams and Operators*. ICDT 2005, LNCS 3363, pp.37-52, 2005.

[14] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. *NiagaraCQ: A scalable continuous query system for internet databases*. ACM SIGMOD pages 379-390, May 2000.

[15] U. Srivastava and J. Widom. *Flexible Time Management in Data Stream Systems*. ACM PODS 2004, June 2004.

[16] S. Babu, U. Srivastava and J. Widom, *Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams*. ACM TODS, Sep. 2004.

[17] Chuck Cranor, Theodore Johnson, Oliver Spatascheck and Vladislav Shkapenyuk, *Gigascope: A Stream Database for Network Applications*. ACM SIGMOD, June 9-12 2003.

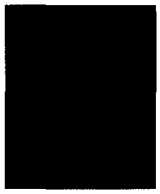
[18] A. Arasu, S. Babu and J. Widom, *The CQL Continuous Query Language: Semantic Foundations and Query Execution*. Stanford University Technical Report, Oct. 2003.

[19] Hyeon Gyu Kim, Cheolgi Kim and Myoung Ho Kim, *Adaptive Disorder Control in Continuous Data Streams*, IEEE CIT, September 2006.

[20] TinyDB: http://www.tinyos.net.

[21] SENSIM: http://csc.lsu.edu/sensor_web/simulator.html.

[22] NS2 Sensor Network Extension: http://pf.itd.nrl.navy.mil/nrlsensorsim



김 현 규

1997년 울산대학교 전산학과 학사. 2000년 울산대학교 전산학과 석사. 2000년~2001년 한국국방연구원 연구원. 2001년~2004년 LG전자 단말연구소 선임연구원. 2005년~현재 한국과학기술원 전산학과 박사과정. 관심분야는 데이타베이스 시스템, 스트림 데이타 처리 등

김 철 기

1996년 한국과학기술원 전산학과 학사 1998년 한국과학기술원 전산학과 석사 2005년 한국과학기술원 전자전산학과 박사. 2001년~2004년 LG전자 단말연구소 선임연구원. 2004년~현재 한국 정보통신대학교 연구원, 연구교수. 관심분야는 이동적응망, 무선인지기술, 센서네트워크 등

김 명 호

정보과학회논문지 : 데이타베이스
제 33 권 제 1 호 참조