www.calsec.or.kr
http://dx.doi.org/10.7838/jsebs.2013.18.3.159

전자상거래 데이터의 실시간 분석을 위한 데이터 스트림과 다수 릴레이션 간의 효율적인 연속 조인 처리 기법

Efficient Processing of Continuous Join Queries between a Data Stream and Multiple Relations for Real-Time Analysis of E-Commerce Data

김해리(Haeri Kim)[*], 이기용(Ki Yong Lee)[**]

초 록

최근 들어 전자상거래 데이터의 실시간 공급이 가능해지면서, 전자상거래 데이터를 실시간으로 분석하고자 하는 요구가 급증하고 있다. 이를 위해서는 전자상거래 데이터 스트림과 디스크에 저장된 대규모 릴레이션 간의 연속 조인 질의를 효율적으로 처리하는 것이 매우 중요하다. 본 논문에서는 전자상거래 데이터 스트림과 디스크에 저장된 다수 릴레이션 간의 효율적인 연속 조인 질의 기법을 제안한다. 제안 방법은 기존 방법에 비해 서비스율을 크게 향상시키는 한편, 메모리 사용량을 크게 줄인다. 분석과 다양한 실험을 통해, 제안 방법은 기존 방법에 비해 서비스율과 메모리 사용량에서 더 효율적임을 보인다.

ABSTRACT

Recently, as real-time availability of e-commerce data becomes possible, the requirement of real-time analysis of e-commerce increases significantly. In the real-time analysis of e-commerce data, it is very important to efficiently process continuous join queries between an e-commerce data stream and disk-based large relations. In this paper, we propose an efficient method for processing a continuous join query between an e-commerce data stream and multiple disk-based relations. The proposed method improves the service rate significantly, while reducing the amount of required memory substantially. Through analysis and various experiments, we show the efficiency of the proposed method compared with the previous one in terms of service rate and memory usage.

키워드 : 연속 조인 질의, 데이터 스트림, 실시간 분석
Continuous Join Query, Data Streams, Real-Time Analysis

---

* M.S. Candidate, Division of Computer Science, Sookmyung Women's University
** Corresponding Author, Assistant Professor, Division of Computer Science, Sookmyung Women's University
(E-mail : kiyonglee@sookmyung.ac)
2013년 07월 17일 접수, 2013년 08월 05일 심사완료 후 2013년 08월 09일 게재확정.

1. 서 론

최근 들어 전자상거래 데이터의 실시간 공급이 가능해지면서, 전자상거래 데이터를 실시간으로 분석하고자 하는 요구가 급증하고 있다. 전자상거래 데이터를 실시간으로 분석하기 위해서는 실시간으로 끊임없이 유입되는 전자상거래 데이터 스트림(data stream)과 디스크에 저장된 대규모 릴레이션(relation) 간의 연속 조인 질의(continuous join query)를 효율적으로 처리하는 것이 매우 중요하다[1, 2, 3].

실시간으로 유입되는 전자상거래 데이터 스트림을 S라 하고, 디스크에 저장된 대규모 릴레이션을 R이라 할 때, S와 R 간의 연속 조인 질의(continuous join query)는 $S \bowtie R$로 표현된다. 그 사용 예로서, 어떤 시각(*timestamp*)에 어떤 제품(*productID*)이 판매되었을 때 그를 나타내는 튜플(tuple) (*timestamp*, *productID*)가 데이터 스트림 S에 실시간으로 유입된다고 하고, 디스크에 저장된 릴레이션 R의 각 튜플(*productID*, *price*)이 각 제품(*productID*)의 가격(*price*)를 나타낸다고 할 때, 연속 조인 질의 $S \bowtie R$의 각 결과 튜플(*timestamp*, *productID*, *price*)는 실시간으로 판매되는 각 제품의 가격을 나타낸다.

이러한 연속 조인 질의 $S \bowtie R$을 처리하기 위해, 전통적인 중첩 루프 조인(nested loop join)[4]이나 색인 기반 조인(index-based join)[4]을 사용하면, S에 도착하는 매 튜플마다 그와 조인되는 R의 튜플을 찾기 위해 디스크를 여러 번 접근해야 하므로 성능이 매우 저하된다. 이를 극복하기 위해 최근에 메시 조인(mesh join)이라는 기법이 제안되었다[5, 6]. 메시 조인은 S에 도착하는 w개의 튜플 당 디스크를 한 번만 접근함으로써 $S \bowtie R$의 처리 속도를 크게 향상시켰다.

하지만 최근 들어 전자상거래 데이터 스트림 S와 디스크에 저장된 다수의 릴레이션 R_1, R_2, …, R_N 간의 연속 다중(multi-way) 조인 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$에 대한 처리가 점차 중요해지고 있다[2]. 예를 들어, 어떤 시각(*timestamp*)에 어떤 제품(*productID*)가 어떤 매장(*storeID*)에서 판매되었을 때 그를 나타내는 튜플(*timestamp*, *productID*, *storeID*)이 데이터 스트림 S에 실시간으로 유입된다고 하자. 디스크에 저장된 릴레이션 R_1의 각 튜플 (*productID*, *price*)는 각 제품(*productID*)의 가격(*price*)을 나타내고, 디스크에 저장된 다른 릴레이션 R_2의 각 튜플(*storetID*, *location*)은 각 매장(*storeID*)의 위치(*location*)를 나타낸다고 할 때, 연속 조인 질의 $S \bowtie R_1 \bowtie R_2$의 각 결과 튜플(*timestamp*, *productID*, *price*, *storeID*, *location*)은 실시간으로 판매되는 각 제품의 가격과 판매 위치를 나타낸다.

그러나 연속 다중 조인 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$을 처리하기 위해 기존의 메시 조인을 단순히 적용하면 문제점이 발생한다. 제 2.1절에서 다시 자세히 설명하겠지만, 메시 조인은 각 릴레이션 R_1, R_2, …, R_N을 고정된 크기의 블록(block)으로 나누고, 각 릴레이션에서 한 블록씩을 메모리에 적재한다. 각 릴레이션 $R_i(i = 1, 2, \cdots N)$의 블록 수를 $B(R_i)$라 하자. 메시 조인은 S에 w개의 튜플이 새로 도착할 때마다, 메모리에 적재된 R_1, R_2, …, R_N의 블록들 중 하나를 그가 속한 릴레이션의 그 다음 블록과 교체한다. 그리고 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot$

$B(R_N)$개의 튜플들과 현재 메모리에 적재된 R_1, R_2, ⋯, R_N의 블록들 간의 조인을 수행하고 그 결과를 출력으로 내보낸다. 따라서 S에 w개의 튜플이 새로 도착할 때마다 디스크를 한 번만 접근하면 된다. 하지만 이렇게 메시 조인을 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$의 처리에 단순하게 적용하면, S에 w개의 튜플이 새로 도착할 때마다 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들과 각 릴레이션 R_1, R_2, ⋯, R_N에서 하나씩 총 N개의 블록들이 모두 참여하는 조인이 매번 수행되어야 한다. 이로 인해 이 조인 수행 시간에 의해 서비스율이 제약을 받게 된다. 더우기 이 방법은 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들을 메모리에 유지하고 있어야 하므로, 블록의 개수가 많아질수록 요구되는 메모리의 양이 매우 커지게 된다.

본 논문에서는 이러한 문제점을 극복하기 위해, 전자상거래 데이터 스트림 S와 디스크에 저장된 다수의 릴레이션 R_1, R_2, ⋯, R_N 간의 연속 다중 조인 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$을 효율적으로 처리하는 기법을 제안한다. 제안 방법은 S에 w개의 튜플이 새로 도착할 때마다, 메모리에 적재된 R_1, R_2, ⋯, R_N의 블록들을 모두 조인에 참여시키는 대신 우선 R_1의 블록과만의 조인을 수행한다. 이후 S에 w개의 튜플이 계속해서 새로 도착하여 비로소 R_1의 블록들과의 조인 결과가 w개가 되면, 그 w개의 조인 결과와 R_2의 블록과의 조인을 수행한다. 이와 같은 방식으로 이후 S에 w개의 튜플이 계속하여 새로 도착하여 비로소 R_{i-1}(i = 2, 3, ⋯, N)의 블록들과의 조인 결과가 w개가 되면, 그 w개의 조인 결과와 R_i의 블록과의 조인을 수행한다. 이와 같은 방식으로 조인을 수행하면, R_N의 블록과의 조인 결과가 최종 질의 결과가 된다. 이렇게 단계적인 방식으로 연속 다중 조인을 처리하면, 메시 조인을 단순히 적용할 때에 비해 S에 w개의 튜플이 새로 도착할 때마다 수행되어야 하는 조인의 비용을 크게 줄일 수 있다. 그에 따라 서비스율이 크게 향상된다. 또한 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들을 메모리에 유지하는 대신, R_1, R_2, ⋯, R_N에 대해 각각 $w \cdot B(R_1)$, $w \cdot B(R_2)$, ⋯, $w \cdot B(R_N)$개의 튜플들씩만 메모리에 유지하면 된다. 따라서 메시 조인을 단순히 적용할 때에 비해 요구되는 메모리양도 크게 줄일 수 있다.

본 논문의 구성은 다음과 같다. 제 2장에서는 관련 연구를 기술하고, 제안 방법의 기반이 되는 메시 조인을 설명한다. 또한 메시 조인을 연속 다중 조인에 단순히 적용했을 때 발생하는 문제점을 기술한다. 제 3장에서는 제안 방법을 설명하고, 제안 방법의 성능을 메시 조인을 단순히 적용했을 때와 비교 분석한다. 제 4장에서는 다양한 실험을 통해 제안 방법의 효율성을 증명한다. 마지막으로 제 5장에서는 결론을 맺는다.

2. 관련 연구

2.1 데이터 스트림과 디스크 기반 릴레이션 간의 연속 조인 질의 처리

데이터 스트림 간의 연속 조인 질의 처리에 대해서는 이미 많은 연구가 이루어져왔다[7,

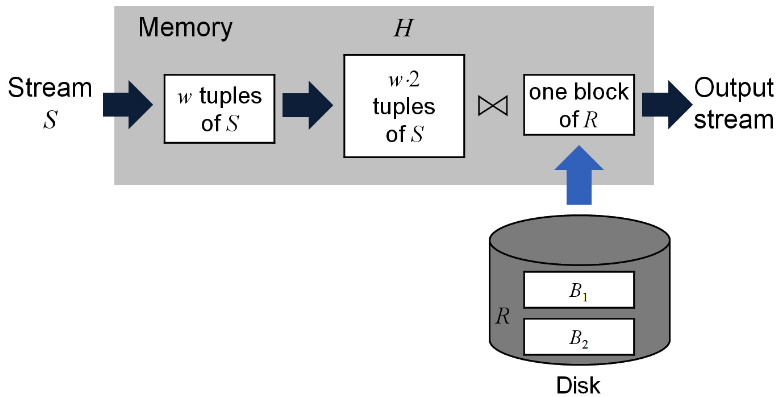
8, 9, 13]. 하지만 데이터 스트림과 디스크에 저장된 릴레이션 간의 연속 조인 질의에 대해서는 아직까지 많은 연구가 이루어지지 않았다. 데이터 스트림과 디스크에 저장된 릴레이션 간의 연속 조인을 처리하는 가장 간단한 방법은 색인 중첩 루프 조인(indexed nested loop join, INLJ)[4]을 사용하는 것이다. INLJ는 데이터 스트림 S와 디스크 기반 릴레이션 R이 있을 때, 먼저 R의 조인 애트리뷰트에 대해 색인을 생성해 놓는다. 이 후 S에 새로운 튜플 t가 도착할 때마다 R에서 t와 조인될 튜플들을 R의 조인 애트리뷰트에 대해 생성된 색인을 사용하여 찾은 뒤, t와 그들을 조인한 결과를 질의 결과로 내보낸다. 하지만 이 방법의 문제점은 S에 들어오는 매 튜플마다 R에서 그와 조인되는 튜플들을 찾기 위해 색인 탐색이 반복적으로 수행된다는 것이다. 일반적으로 색인 탐색은 디스크에 대한 여러 번의 임의 접근(random access)을 발생시키기 때문에, 반복적인 색인 탐색은 디스크 I/O 비용을 매우 크게 증가시킨다. 따라서 S에 매우 빠른 속도로 튜플들이 계속해서 들어오게 되면 질의 처리 속도가 크게 저하된다. 또한 R에 대해 B+-tree와 같은 색인을 유지하는 데 드는 비용도 문제가 된다. 이러한 INLJ의 문제점을 극복하기 위해 최근에 메시 조인이라는 방법이 제안되었다[5, 6]. 다음 절에서는 본 논문에서 제안하는 방법의 기반이 되는 메시 조인을 자세히 설명한다.

2.2 메시 조인(Mesh Join)

본 절에서는 데이터 스트림 S와 하나의 디스크 기반 릴레이션 R 간의 연속 조인 질의 $S \bowtie R$를 효율적으로 처리하기 위해 제안된 메시 조인을 설명한다. $S \bowtie R$을 처리하기 위해서는 S에 도착하는 각 튜플에 대해 R에서 그와 조인되는 튜플을 찾아 그들 간의 조인 결과를 출력으로 내보내야 한다.

메시 조인은 R을 동일한 크기의 블록 B_1, B_2, …, B_m으로 나눈다(m은 블록의 개수). 메시 조인은 S에 새로운 튜플이 도착할 때 마다 각각을 개별적으로 처리하는 것이 아니라, w개의 튜플을 모아 이들을 일괄적으로 처리한다. 한편 메시 조인은 S에 가장 최근에 도착한 $w \cdot m$개의 튜플들을 메모리 저장공간 H에 유지하고 있다. S에 w개의 새로운 튜플이 도착하면, 메시 조인은 새로 도착한 w개의 튜플을 H에 추가하고, 가장 오래된 w개의 튜플을 H로부터 제거한다. 그리고 R의 한 블록 $B_i(1 \leq i \leq m)$을 메모리에 적재한다. S에 w개의 새로운 튜플이 도착할 때마다, R의 블록들은 B_1, B_2, …, B_m순으로 하나씩 순차적으로 메모리에 적재된다. B_m 다음에는 다시 B_1 차례가 된다. 그 후 메시 조인은 H에 저장되어 있는 S의 튜플들(즉, S에 가장 최근에 도착한 $w \cdot m$개의 튜플들)과 현재 메모리에 적재된 R의 블록 B_i 간의 조인을 수행하고, 그 결과를 출력으로 내보낸다.

<Figure 1>은 메시 조인의 수행 예를 보여준다. R은 두 개의 블록 B_1, B_2로 나뉘어 있으며, 메모리에는 S에 가장 최근에 도착한 $w \cdot 2$개의 튜플을 저장할 저장공간 H가 있다. S에 w개의 튜플이 처음으로 새로 도착하면, 이들을 H에 추가한다. 그리고 R의 첫 블록 B_1을 메모리에 적재하여 H와 B_1에 포함된 튜플 간의 조인을 수행하고, 그 결과를 출력으로 내보낸다. 그 후 S에 w개의 튜플이 새



〈Figure 1〉 Mesh Join

로 또 도착하면, 이들을 H에 추가한다. 그리고 R의 다음 블록 B_2를 메모리에 적재하여 H와 B_2에 포함된 튜플 간의 조인을 수행하고, 그 결과를 출력으로 내보낸다. 그 후 S에 w개의 튜플이 새로 또 도착하면, 이들을 H에 새로 추가하고, 가장 오래된 w개의 튜플을 H에서 제거한다. 그리고 R의 첫 블록 B_1을 다시 메모리에 적재하여 H와 B_1에 포함된 튜플 간의 조인을 수행하고, 그 결과를 출력으로 내보낸다. 이렇게 조인을 수행하면 S에 도착한 각 튜플들은 H에 추가되어 H에서 제거되기 전까지 R의 모든 블록 B_1, B_2와 한 번씩 조인이 수행됨이 보장된다. 따라서 S에 w개의 튜플이 새로 도착할 때마다 R의 블록을 한 번씩만 읽으면 되므로 디스크에 접근하는 비용이 크게 줄어든다. 그 결과 $S \bowtie R$를 매우 효율적으로 처리할 수 있다.

최근에는 메시 조인의 성능을 더욱 높이기 위한 몇 가지 방법들이 더 제안되었다. R-MESHJOIN은 동일한 양의 메모리를 사용할 때, R의 블록 수와 H의 튜플 수를 최적화하여 R에 대한 디스크 접근 비용을 더욱 감소시키는 방법을 제안하였다[10]. X-hybrid join은 R의 튜플들 중 조인 결과에 자주 참여하는 튜플들을 메모리에 더 오랫동안 유지함으로써, R에 대한 디스크 접근 비용을 더욱 줄이는 방법을 제안하였다[11]. 최근에는 X-hybrid join의 성능을 더욱 개선한 Optimized X-hybrid join이 제안되었다[12]. 하지만 메시 조인을 포함하여 이들 연구는 모두 데이터 스트림 S와 단일 릴레이션 R과의 연속 조인 질의 $S \bowtie R$에 대한 것이며, S와 다수의 릴레이션 $R_1, R_2, \cdots, R_N$ 간의 연속 조인 질의 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$에 대해서는 언급하지 않고 있다.

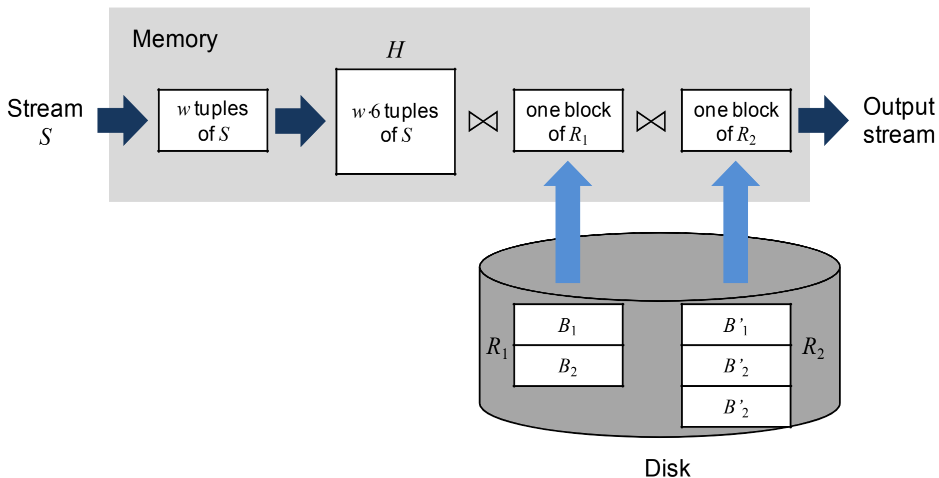
2.3 메시 조인을 연속 다중 조인에 단순히 적용했을 때의 문제점

제 2.2절에서 설명한 메시 조인은 데이터 스트림 S와 단일 릴레이션 R 간의 연속 조인 $S \bowtie R$을 처리하기 위해 제안된 기법이다. 하지만 아직까지 본 논문에서 다루는 문제인 연속 다중 조인 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$을 위해 특별히 제안된 기법은 찾지 못하였다. 이에 따라 본 논문에서는 제안 방법을, 메시 조인을 연속 다중 조인 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$의

처리에 단순히 적용했을 때와 비교한다. 메시 조인을 연속 다중 조인 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$의 처리에 단순히 적용하면 다음과 같이 된다.

각 릴레이션 R_i의 블록 수를 $B(R_i)$라 하자. 메시 조인으로 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$을 처리하기 위해서는 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들을 메모리 저장공간 H에 유지하고 있어야 한다. 또한 각 릴레이션 R_i에 대해 그의 블록을 하나씩 저장할 수 있는 저장공간이 필요하다. S에 w개의 튜플이 새로 도착하면, 새로 도착한 w개의 튜플을 H에 추가하고, 가장 오래된 w개의 튜플을 H로부터 제거한다. 그리고 $R_1, R_2, \cdots, R_N$ 중에서 하나를 선택하여 현재 메모리에 적재된 그의 블록을 제거하고 그의 다음 블록을 메모리에 적재한다. S에 w개의 튜플이 새로 도착할 때마다, $R_1, R_2, \cdots, R_N$의 블록들은 각 R_i에서 한 블록씩 뽑아 만들 수 있는 $B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 모든 가능한 블록들의 조합이 한 번씩 나타나도록 하나씩 교체되어 메모리에 적재된다. 그 후 메시 조인은 H에 저장되어 있는 S의 튜플들(즉, S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들)과 현재 메모리에 적재된 $R_1, R_2, \cdots, R_N$의 블록들 간의 조인을 수행하고, 그 결과를 출력으로 내보낸다. 이렇게 조인을 수행하면 S에 도착한 각 튜플들은 H에 추가되어 H에서 제거되기 전까지 $R_1, R_2, \cdots, R_N$에서 한 블록씩 뽑아 만들 수 있는 모든 가능한 블록들의 조합과 한 번씩 조인이 수행됨이 보장된다.

<Figure 2>는 메시 조인을 $S \bowtie R_1 \bowtie R_2$의 처리에 단순히 적용시킨 예를 보여준다. R_1은 두 개의 블록 B_1, B_2로, R_2는 세 개의 블록 B'_1, B'_2, B'_3로 나뉘어져 있다. 메모리에는 S에 가장 최근에 도착한 $w \cdot 2 \cdot 3 = w \cdot 6$개의 튜플을 저장할 저장공간 H가 있다. S에 w개의 튜플이 새로 도착할 때마다, 새로 도착한 튜플을 H에 추가하고, 가장 오래된 w개의 튜플을 H로부터 제거한다. 그리고 S에 w개의 튜플이 새로 도착할 때마다, R_1과



〈Figure 2〉 A Naive Extension of Mesh Join to Processing $S \bowtie R_1 \bowtie R_2$

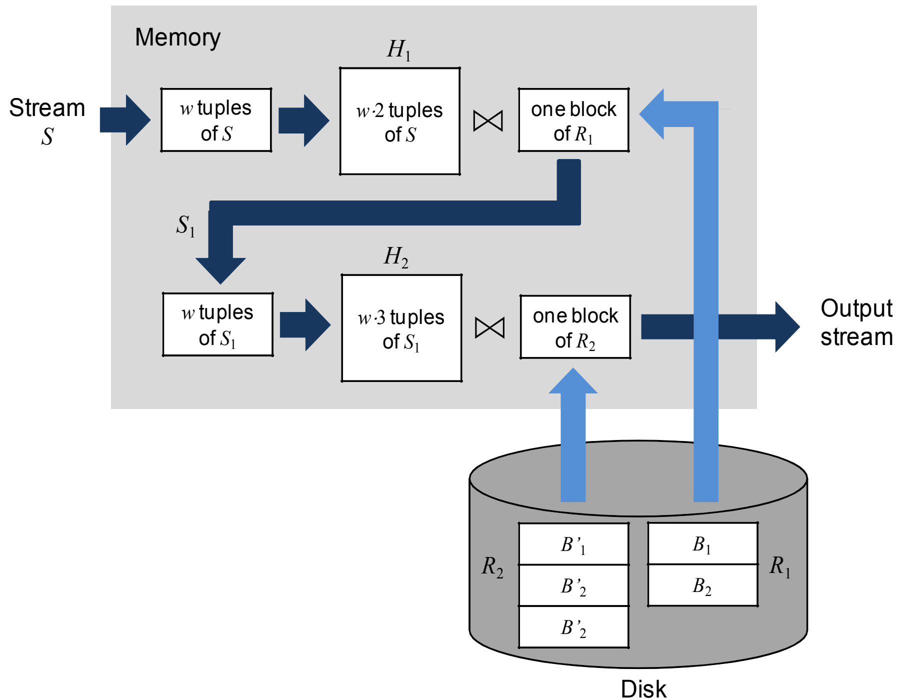
R_2에서 각각 한 블록씩 뽑아 만들 수 있는 총 6개의 조합(즉, (B_1, B'_1), (B_1, B'_2), (B_1, B'_3), (B_2, B'_3), (B_2, B'_2), (B_2, B'_1))이 한 번씩 나타나도록 R_1과 R_2의 블록들 중 한 블록씩을 교체하여 메모리에 적재한다. 그 후 메시 조인은 H에 저장되어 있는 S의 튜플들(즉, S에 가장 최근에 도착한 $w \cdot 6$개의 튜플들)과 현재 메모리에 적재된 R_1과 R_2의 블록들 간의 조인을 수행하고, 그 결과를 출력으로 내보낸다. 이렇게 조인을 수행하면 S에 도착한 각 튜플들은 H에 추가되어 H에서 제거되기 전까지 R_1과 R_2에서 한 블록씩 뽑아 만들 수 있는 모든 가능한 블록들의 조합과 한 번씩 조인이 수행됨이 보장된다. 따라서 S에 w개의 튜플이 새로 도착할 때마다 R_1과 R_2의 블록들 중 한 블록씩만 읽어 $S \bowtie R_1 \bowtie R_2$를 처리하는 것이 가능하다.

하지만 이 방법은 다음과 같은 문제점을 가진다. (1) 반복되는 조인 연산의 비용 : S에 w개의 튜플이 새로 도착할 때마다, H에 저장된 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들과 각 릴레이션 R_1, R_2, ⋯, R_N에서 하나씩 총 N개의 블록들이 매번 모두 조인 연산에 참여해야 한다. 이러한 조인 연산은 결과 튜플이 전혀 생성되지 않는 경우에도 무조건 수행된다. 이 조인 연산은 S에 w개의 튜플이 새로 도착할 때마다 매번 수행되므로, 이 조인 연산에 드는 시간에 의해 서비스율이 제약을 받게 된다. (2) 메모리 과다 사용 : H에 저장된 S의 튜플들은 R_1, R_2, ⋯, R_N에서 한 블록씩 뽑아 만들 수 있는 모든 가능한 블록들의 조합과 한 번씩 조인이 수행되기 전까지는 H에서 제거될 수 없다. 이에 따라 H는 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들을 저장하고 있어야 한다. 따라서 릴레이션의 크기가 커지거나 릴레이션의 개수가 많아져서 블록 수가 증가할수록 요구되는 메모리량이 매우 커지게 된다.

3. 제안 방법

3.1 개요

제안 방법은 S에 w개의 튜플이 새로 도착할 때마다, 메모리에 적재된 R_1, R_2, ⋯, R_N의 블록들을 모두 조인에 참여시키는 대신 R_1, R_2, ⋯, R_N과의 조인을 단계적으로 수행한다. 제안 방법은 S에 w개의 튜플이 새로 도착하면, 제 2.2절에서 설명한 메시 조인을 사용하여 S와 R_1 간의 조인을 수행하고 그의 결과를 스트림 S_1으로 내보낸다. 이후 S에 w개의 튜플이 계속해서 새로 도착하여 비로소 S_1에 새로 도착한 튜플이 w개가 되면, 제안 방법은 S에 w개의 튜플이 새로 도착했을 때와 마찬가지로 메시 조인을 사용하여 S_1과 R_2 간의 조인을 수행하고 그의 결과를 스트림 S_2로 내보낸다. 이후 S에 w개의 튜플이 계속해서 새로 도착하여 비로소 S_2에 새로 도착한 튜플이 w개가 되면, 제안 방법은 메시 조인을 사용하여 S_2과 R_3 간의 조인을 수행하고 그의 결과를 스트림 S_3로 내보낸다. 이와 같은 방식으로 하여, 이후 S에 w개의 튜플이 계속하여 새로 도착하여 S_{N-1}과 R_N 간의 조인이 수행되면 그의 결과가 최종 질의 결과가 된다.



<Figure 3> Example of the Proposed Method

<Figure 3>는 제안 방법으로 연속 다중 질의 $S \bowtie R_1 \bowtie R_2$를 처리하는 예를 보여준다. <Figure 2>에서와 마찬가지로 R_1은 두 개의 블록 B_1, B_2로, R_2는 세 개의 블록 B'_1, B'_2, B'_3로 이루어져 있다. 메모리에는 S에 가장 최근에 도착한 $w \cdot 2$개의 튜플을 저장할 저장공간 H_1이 있다. S에 w개의 튜플이 새로 도착하면, 메시 조인과 같이 S에 새로 도착한 튜플을 H_1에 추가하고, 가장 오래된 w개의 튜플을 H_1로부터 제거한다. 그리고 현재 메모리에 적재된 R_1의 블록을 그의 다음 블록으로 교체한다. 그 후 H_1과 현재 메모리에 적재된 R_1의 블록에 포함된 튜플 간의 조인을 수행하고, 그 결과를 데이터 스트림 S_1으로 내보낸다. 따라서, S_1은 연속 조인 질의 $S \bowtie R_1$의 결과를 나타내는 스트림으로 볼 수 있다. 또한 메모리에는 S_1에 가장 최근에 도착한 $w \cdot 3$개의 튜플을 저장할 저장공간 H_2가 있다. 이후 S에 w개의 튜플이 계속해서 새로 도착하여 비로소 S_1에 새로 도착한 튜플이 w개가 되면, 메시 조인과 같이 S_1에 새로 도착한 튜플을 H_2에 추가하고, 가장 오래된 w개의 튜플을 H_2로부터 제거한다. 그리고 현재 메모리에 적재된 R_2의 블록을 그의 다음 블록으로 교체한다. 그 후 H_2와 현재 메모리에 적재된 R_2의 블록에 포함된 튜플 간의 조인을 수행하고, 그 결과를 최종 질의 결과로 내보낸다. 이렇게 조인을 수행하면 S에 도착한 각 튜플들은 R_1의 모든 블록 B_1, B_2와 각각 한 번씩 조인이 수행됨이 보장되며, S_1에 도착한 각 튜플들(즉, $S \bowtie R_1$의 결과에 포함된 각 튜플들)은 R_2의

모든 블록 B'_1, B'_2, B'_3와 각각 한 번씩 조인이 수행됨이 보장된다. 따라서 $S \bowtie R_1 \bowtie R_2$의 결과가 올바르게 출력됨이 보장된다.

이렇게 단계적인 방식으로 연속 다중 조인을 처리하면, 메시 조인을 단순히 적용할 때에 비해 S에 w개의 튜플이 새로 도착할 때마다 수행되어야 하는 조인의 비용을 크게 줄일 수 있다. 즉, S에 w개의 튜플이 새로 도착할 때마다 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들과 각 릴레이션 R_1, R_2, $\cdots$, R_N에서 하나씩 총 N개의 블록들을 매번 모두 조인하는 대신, S에 가장 최근에 도착한 $w \cdot B(R_1)$개의 튜플들과 R_1의 한 블록 간의 조인만 수행하면 된다. 그 결과로 만약 $S_i(i = 1, 2, \cdots, N-1)$에 새로 도착한 튜플의 개수가 w개가 되는 경우에만 비로소 S_i에 가장 최근에 도착한 $w \cdot B(R_{i+1})$개의 튜플들과 R_{i+1}의 한 블록 간의 조인이 수행된다. 따라서 S에 w개의 튜플이 새로 도착할 때마다 수행되어야 하는 조인의 비용이 크게 감소되어 서비스율이 크게 향상된다. 또한 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플들을 메모리에 유지하는 대신, R_1, R_2, $\cdots$, R_N에 대해 각각 $w \cdot B(R_1)$, $w \cdot B(R_2)$, $\cdots$, $w \cdot B(R_N)$개씩 총 $w \cdot B(R_1) + w \cdot B(R_2) + \cdots + w \cdot B(R_N)$개의 튜플들만 메모리에 유지하면 된다. 이에 따라 메시 조인을 단순히 적용할 때에 비해 요구되는 메모리양도 크게 줄어든다.

3.2 제안 방법의 분석

본 절에서는 연속 다중 조인 $S \bowtie R_1 \bowtie \cdots \bowtie R_N$의 처리에 메시 조인을 단순하게 적용한 방법(제 2.2절에서 설명)과 제안 방법(제 3.1절에서 설명)을 비교 분석한다. 이를 위해 두 방법의 (1)서비스율과 (2)메모리 사용량을 비교한다. 편의를 위해 제안 방법을 PL(Pipelined Join), 제 2.2절에서 설명한 방법을 MESH로 각각 표기한다. <Table 1>은 아래 분석에서 사용할 기호들을 정리한 것이다. <Table 1>에서 $\sigma(R_i)$는 R_i의 전체 튜플 중에서 S의 튜플과 조인이 이루어지는 튜플들의 비율을 나타내며, [0, 1]사이의 값을 가진다.

〈Table 1〉 Notations for the Cost Model

Notation	Meaning
N	Number of disk-based relations
C_{remove}	Cost of removing a tuple from H
C_{insert}	Cost of inserting a tuple to H
C_{read}	Cost of reading a block from a disk-based relation
C_{hash}	Cost of probing a hash table H
B	Size of a block(in bytes)
$v(R_i)$	Size of a tuple of R_i(in bytes)
$v(S_i)$	Size of a tuple of S_i(in bytes)
$\sigma(R_i)$	The selectivity of R_i
f	The fudge factor of the hash table implementation

먼저 두 방법의 서비스율을 비교한다. 서비스율은 연속 조인 질의 처리 기법에 의해 초당 처리되는 데이터 스트림 S의 튜플 수를 의미한다. MESH에서 S에 새로 도착한 w개의 튜플을 처리하는데 걸리는 시간을 C_{MESH}라 하면 MESH의 서비스율은 $\mu_{MESH} = w/C_{MESH}$로 나타낼 수 있다. 여기서 C_{MESH}는 다음과 같이 각 세부 연산에 걸리는 시간의 총합으로 나타낼 수 있다.

$$C_{MESH} = w \cdot C_{insert} + w \cdot C_{remove} + C_{read} + C_{hash} \cdot \sum_{i=1}^{N} \frac{B}{v(R_i)}$$

첫 번째 항($w \cdot C_{insert}$)은 S에 새로 도착한 w개의 튜플을 H에 추가하는 비용이다. 두 번째 항($w \cdot C_{remove}$)는 가장 오래된 w개의 튜플을 H로부터 제거하는 비용이다. 세 번째 항(C_{read})는 R_1, R_2, …, R_N의 블록들 중 하나를 디스크로부터 읽어 메모리에 적재하는 비용이다. 마지막 네 번째 항은 H에 저장되어 있는 튜플들과 현재 메모리에 적재된 R_1, R_2, …, R_N의 블록들 간의 조인 비용을 나타낸다. 본 논문에서는 이 조인 연산을 빠르게 수행하기 위해, H를 그에 저장되어 있는 튜플들이 조인 애트리뷰트 값으로 해시되어 있는 해시 테이블의 형태로 유지하며, H의 튜플들과 현재 메모리에 적재된 R_1, R_2, …, R_N의 블록들 간의 조인을 해시 조인으로 처리한다. 이 경우 메모리에 적재된 각 R_i의 블록에는 $B/v(R_i)$개 만큼의 튜플이 있으므로 이들을 각각 해시하여 조인하는 비용은 $B/v(R_1) + B/v(R_2) + \cdots + B/v(R_N)$에 비례한다.

반면에 PL에서 S에 새로 도착한 w개의 튜플을 처리하는데 걸리는 시간을 C_{PL}이라 하면, PL의 서비스율은 $\mu_{PL} = w/C_{PL}$로 나타낼 수 있다. 여기서 C_{PL}은 다음과 같다.

$$\begin{aligned} C_{PL} = {} & w \cdot C_{insert} \cdot (1+\sum_{i=2}^{N}\prod_{j=1}^{i-1}\sigma(R_j)) \\ & + w \cdot C_{remove} \cdot (1+\sum_{i=2}^{N}\prod_{j=1}^{i-1}\sigma(R_j)) \\ & + C_{read} \cdot (1+\sum_{i=2}^{N}\prod_{j=1}^{i-1}\sigma(R_j)) \\ & + C_{hash} \cdot (\frac{B}{v(R_1)}+\sum_{i=2}^{N}\frac{B}{v(R_i)} \cdot \prod_{j=1}^{i-1}\sigma(R_j)) \end{aligned}$$

첫 번째 항은 S, S_1, S_2, …, S_{N-1} 각각에 새로 도착한 튜플들을 각각 H_1, H_2, …, H_N에 추가하는 비용이다. S에 w개의 튜플이 새로 도착하면, H_1에는 w개의 튜플이 추가된다. 그 결과 H_1과 R_1의 한 블록이 조인되면, H_1에는 $w \cdot B(R_1)$개의 튜플이 있고, R_1에서는 총 $B(R_1)$개의 블록 중 하나가 조인에 참여하는 것이므로 평균적으로 $w \cdot B(R_1) \cdot (1/B(R_1)) \cdot \sigma(R_1) = w \cdot \sigma(R_1)$개의 튜플이 S_1에 새로 도착한다. 결국 H_2에는 평균적으로 $w \cdot \sigma(R_1)$개의 튜플이 추가된다. 이와 같이 계산하면, S에 w개의 튜플이 새로 도착할 때마다 각 H_i(i = 2, 3, …, N)에는 평균적으로 $w \cdot \sigma(R_1) \cdot \sigma(R_2) \cdot \cdots \cdot \sigma(R_{i-1})$개의 튜플이 추가된다. 따라서 H_1, H_2, …, H_N에 새로운 튜플들을 추가하는 비용을 다 더하면 첫 번째 항과 같이 된다. 두 번째 항은 각 H_1, H_2, …, H_N에서 추가된 튜플의 개수만큼 가장 오래된 튜플을 제거하는 비용이며, 첫 번째 항과 유사한 방법으로 계산할 수 있다. 세 번째 항은 R_1, R_2, …, R_N 각각에 대해 다음 블록을 디스크에서 읽어 메모리에 적재하는 비용이다. S에 w개의 튜플이 도착하면, R_1에서는 한 개의 블록을 읽어 메모리에 적재한다. 하지만 이 때 S_1에는 평균적으

로 $w \cdot \sigma(R_1)$개의 튜플이 도착하므로, R_2에서는 평균적으로 $\sigma(R_1)$개의 블록을 읽는다. 이와 같이 계산하면, $R_i(i = 2, 3, \cdots, N)$에서는 평균적으로 $\sigma(R_1) \cdot \sigma(R_2) \cdot \cdots \cdot \sigma(R_{i-1})$개의 블록을 읽는다. 따라서 $R_1, R_2, \cdots, R_N$ 각각에 대해 다음 블록을 디스크에서 읽어 메모리에 적재하는 비용을 다 더하면 세 번째 항과 같이 된다. 마지막 네 번째 항은 $H_1, H_2, \cdots, H_N$와 현재 메모리에 적재된 $R_1, R_2, \cdots, R_N$의 한 블록씩을 각각 조인하는 비용의 총합이다. 위에서 설명한대로 해시 조인을 사용하면, H_i와 R_i의 한 블록을 조인하는 비용은 $C_{hash} \cdot (B/v(R_i))$으로 나타낼 수 있다. 하지만 H_i와 R_i의 한 블록 간의 조인은 S_{i-1}에 새로 도착한 튜플이 w개가 되었을 때만 한 번씩 수행되는 것이고, S에 w개의 튜플이 도착했을 때 S_{i-1}에 새로 도착하는 튜플의 수는 평균적으로 $w \cdot \sigma(R_1) \cdot \sigma(R_2) \cdot \cdots \cdot \sigma(R_{i-1})$이므로, S에 w개의 튜플이 새로 도착했을 때 발생하는 H_i와 R_i의 한 블록 간의 조인 비용은 평균적으로 $C_{hash} \cdot (B/v(R_i)) \cdot \sigma(R_1) \cdot \sigma(R_2) \cdot \cdots \cdot \sigma(R_{i-1})$이 된다. 따라서 이를 다 더하면 네 번째 항과 같이 된다.

앞서 분석한 C_{MESH}와 C_{PL}을 비교하면, PL에서 $H_1, H_2, \cdots, H_N$에 튜플을 추가하고 제거하는 비용의 합은 MESH에서 H에 튜플을 추가하고 제거하는 비용에 비해 비교적 크다. 또한 PL은 MESH에 비해 $R_1, R_2, \cdots, R_N$에서 블록을 읽어들이는 비용이 더 크다. 하지만 이 비용은 제 4장에서 보일 바와 같이 전체 비용에서 매우 작은 부분을 차지하며, 대부분은 조인을 수행하는 비용이 차지한다. 더우기 이 비용은 $\sigma(R_1), \sigma(R_2), \cdots, \sigma(R_N)$의 값이 작아질수록 PL과 MESH 간의 차이가 급격히 줄어든다. 즉, C_{PL}의 식에서 $1+\sum_{i=2}^{N}\prod_{j=1}^{i-1}\sigma(R_j) \approx 1$이 된다. 특히 전체 비용에서 대부분을 차지하는, S에 w개의 튜플이 도착할 때마다 수행되는 조인의 비용을 비교하면, MESH에서는 매번 모든 R_i에서 항상 $B/v(R_i)$개의 튜플들이 조인에 참여한다. 반면에 PL에서는 R_1을 제외한 각 $R_i(i = 2, 3, \cdots, N)$에서 평균적으로 $(B/v(R_i)) \cdot \sigma(R_1) \cdot \sigma(R_2) \cdot \cdots \cdot \sigma(R_{i-1})$개 만큼의 튜플들만이 조인에 참여한다. 따라서 PL은 MESH에 비해 더 적은 비용으로 조인을 수행할 수 있다.

다음은 두 방법의 메모리 사용량을 비교한다. 먼저 MESH의 메모리 사용량 M_{MESH}는 다음과 같다.

$$M_{MESH} = N \cdot B + w \cdot v(S) + f \cdot w \cdot v(S) \cdot \prod_{i=1}^{N} B(R_i)$$

첫 번째 항($N \cdot B$)은 각 릴레이션 $R_1, R_2, \cdots, R_N$의 블록을 하나씩 적재하는데 필요한 메모리 사용량이다. 두 번째 항($w \cdot v(S)$)은 S에 가장 최근에 도착한 w개의 튜플을 저장하는데 필요한 버퍼의 크기를 나타낸다. 마지막 항은 S에 가장 최근에 도착한 $w \cdot B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$개의 튜플을 저장하는 메모리 저장공간 H의 크기를 나타낸다(f는 해시 테이블 구현 방법에 따라 튜플 하나 당 필요한 실제 공간의 비를 나타내는 상수이다).

반면에 PL의 메모리 사용량 M_{PL}는 다음과 같다.

$$M_{PL} = N \cdot B + w \cdot (v(S) + \sum_{i=1}^{N-1} v(S_i)) + f \cdot w \cdot (v(S) \cdot B(R_1) + \sum_{i=1}^{N-1} v(S_i) \cdot B(R_{i+1}))$$

첫 번째 항($N \cdot B$)은 각 릴레이션 R_1, R_2, ⋯, R_N의 블록을 하나씩 적재하는데 필요한 메모리 사용량이다. 두 번째 항은 데이터 스트림 S, S_1, ⋯, S_{N-1} 각각에 대해 가장 최근에 도착한 w개의 튜플을 저장하는데 필요한 버퍼 크기의 총합을 나타낸다. 마지막 항은 저장공간 H_1, H_2, ⋯, H_N의 크기의 총합을 나타낸다.

따라서 PL은 MESH에 비해 S_1, S_2, ⋯, S_{N-1} 각각에 대해 가장 최근에 도착한 w개의 튜플을 저장하기 위한 버퍼가 부가적으로 필요하다. 하지만 MESH에서는 H가 $B(R_1) \cdot B(R_2) \cdot \cdots \cdot B(R_N)$에 비례하는 메모리를 사용하는 반면, PL는 H_1, H_2, ⋯, H_N이 $B(R_1) + B(R_2) + \cdots + B(R_N)$에 비례하는 저장공간만을 사용한다. 이에 따라 총 메모리 사용량을 크게 줄임을 알 수 있다.

4. 성능 평가

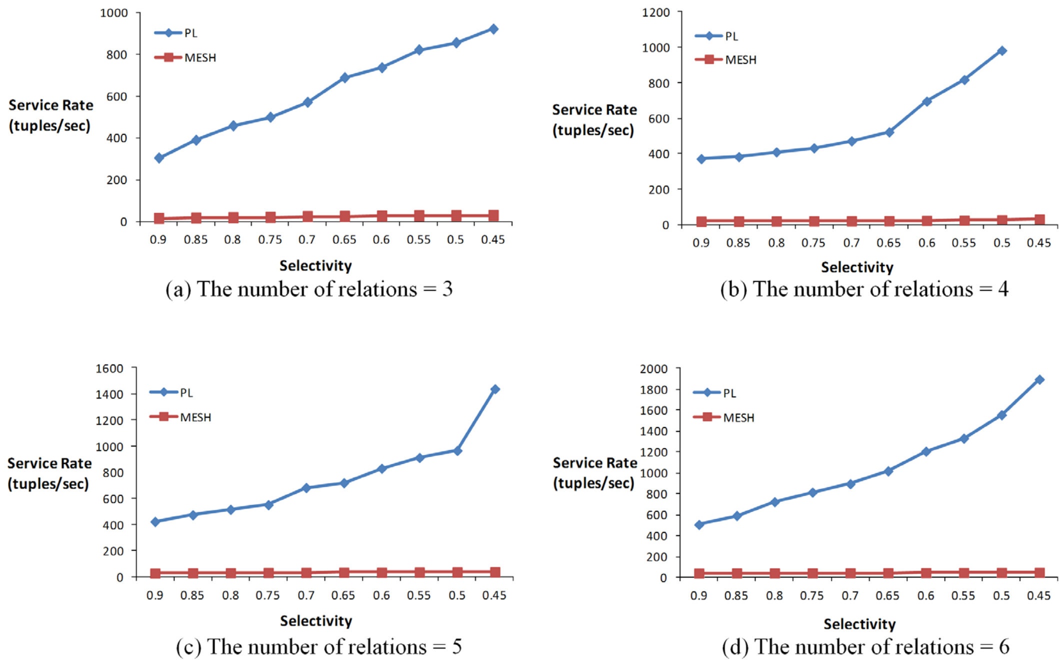
4.1 실험 환경 및 방법

본 절에서는 PL와 MESH의 성능을 비교하기 위한 실험 환경 및 방법을 기술한다. 두 방법의 성능을 비교하기 위해 사용한 데이터는 다음과 같다. 데이터 스트림과 조인되는 디스크 기반 릴레이션의 개수는 3, 4, 5, 6개로 증가시켜가며 실험을 수행하였다. 또한 주어진 릴레이션의 개수에 대해, 각 릴레이션의 조인 선택도 또는 블록 수를 변화시켜가며 실험을 수행하였다. 6개 릴레이션의 조인 애트리뷰트 도메인은 각각 $[1, 7.2 \times 10^5]$, $[1, 3.0 \times 10^5]$, $[1, 4.8 \times 10^5]$, $[1, 5.0 \times 10^5]$, $[1, 6.0 \times 10^5]$으로 하였으며, 각각 균등 분포(uniform distribution)로 값을 생성하였다. 모든 릴레이션의 튜플 하나의 크기는 400bytes로 동일하게 하였으며, 한 블록에는 2000개의 튜플이 들어가도록 하였다. 데이터 스트림에 도착하는 튜플의 조인 애트리뷰트 값은 주어진 각 릴레이션의 선택도(selectivity)에 맞게 생성하였다. 실험은 Windows 7 운영체제 환경에서 CPU가 Intel Core i7-2600이고 메모리가 4GB인 컴퓨터에서 수행되었다.

본 실험에서는 PL와 MESH의 성능을 서비스율과 메모리 사용량 두 가지 척도로 측정하였다. 서비스율은 초당 처리된 데이터 스트림의 튜플 수로써, 각 방법을 10분간 수행하고 첫 1분 예열 시간을 제외한 나머지 9분 동안 처리된 데이트 스트림 튜플 수를 총 수행 시간(9분)으로 나누어 구하였다. 메모리 사용량은 각 방법이 수행될 때 사용한 메모리의 총량으로서, Windows 작업관리자의 리소스 모니터를 사용하여 직접 측정하였다. 각 척도에 대해 릴레이션의 개수가 각각 3, 4, 5, 6개일 때, 각 릴레이션의 조인 선택도 또는 블록 수를 변화시켜가면서 실험을 수행하였다.

4.2 실험 결과

<Figure 4>는 릴레이션의 개수가 각각 3, 4, 5, 6개일 때 각 릴레이션의 조인 선택도를 변화시켜가면서 PL과 MESH의 서비스율을 측정한 결과이다. 가로축은 각 릴레이션들의 조인 선택도를 나타내며, 세로축은 초당 처리된 데이터 스트림의 튜플 수(tuples/sec)를 나타낸다.



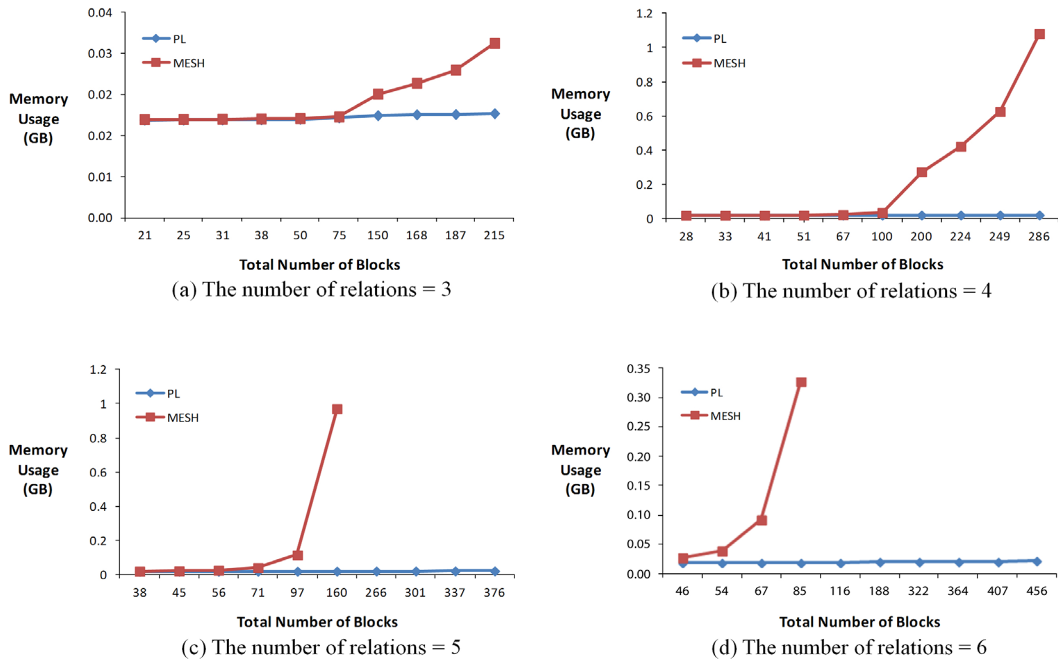
(a) The number of relations = 3 (b) The number of relations = 4

(c) The number of relations = 5 (d) The number of relations = 6

〈Figure 4〉 Service Rate for Varying Selectivity

<Figure 4>에서 볼 수 있듯이 PL은 모든 경우에 대해 MESH보다 훨씬 높은 서비스 처리율을 보이고 있다. 이것은 제 3.2절에서 분석한 것처럼, PL은 MESH에 비해 S에 w개의 튜플이 도착할 때마다 수행되는 조인 연산의 비용이 더 적기 때문이다. 또한 조인 선택도가 감소할수록 서비스 처리율이 증가하는 것을 볼 수 있다. 이것은 조인 선택도가 감소할수록 S에 w개의 튜플이 도착했을 때마다 수행되는 조인 연산의 비용이 줄어들기 때문이다. 즉, PL에서는 조인 선택도가 감소할수록 $S \bowtie R_1 \bowtie \cdots \bowtie R_i$의 수행 결과 다음 S_i로 유입되는 결과 튜플의 수가 큰 폭으로 감소하여, R_2, R_3, …, R_N의 블록이 조인에 참여하는 빈도 수가 크게 줄어든다. 따라서 조인 선택도가 감소할수록 더 많은 입력 스트림 튜플을 처리할 수 있게 된다. MESH에서도 조인 선택도가 감소할수록 H와 R_1, R_2, …, R_N의 한 블록씩 간의 조인 결과를 만들어 내는 비용이 줄어들어 전체적인 조인 비용이 감소한다. 하지만 그에 따른 감소 폭은 PL에 비해 매우 작다.

<Figure 5>는 릴레이션의 개수가 각각 3, 4, 5, 6개일 때, 각 릴레이션의 블록 개수를 증가시키면서 PL과 MESH의 메모리 사용량을 측정한 결과이다. 가로축은 릴레이션들의 블록 수를 모두 더한 값을 나타내며, 세로축은 총 메모리 사용량(GB)를 나타낸다. 6개 릴레이션의 초기 블록 개수는 각각 10, 4, 7, 7, 10, 8개로 하였으며, 동일한 비율을 유지하며 각각을 증가시켰다. 제 3.2절에서 분석한 바와 같이, 두 방법 모두 블록 수가 증가할수록 메모리 사용량이 증가하는 것을 볼 수 있다. 다만 MESH의 메모리 사용량은 각 릴레이



〈Figure 5〉 Memory Usage for Varying Number of Blocks

션의 블록 수의 곱에 비례하여 매우 빠르게 증가하는 반면, PL의 메모리 사용량은 각 릴레이션의 블록 수의 합에 비례하여 증가하기 때문에 상대적으로 거의 변화하지 않는 것처럼 보인다. 특히 릴레이션의 개수가 5개이고 블록 수의 총합이 266개 이상일 때와 릴레이션의 개수가 6개이고 블록 수의 총합이 116개 이상일 때는 MESH에서 메모리 오버플로우가 발생하여 수행이 불가능하였다. 이로부터 제안 방법은 기존 방법에 비해 서비스율을 향상시키는 것 외에도 메모리 사용량을 큰 폭으로 감소시킴을 알 수 있다.

5. 결 론

본 논문은 실시간으로 유입되는 전자상거래 데이터 스트림과 다수의 디스크 기반 릴레이션 간의 연속 다중 조인을 효율적으로 처리하는 기법을 제안하였다. 제안하는 방법은 데이터 스트림에 튜플들이 도착할 때마다 수행되어야 하는 조인 연산의 비용을 줄여 서비스율을 향상시켰다. 또한 메모리에 저장하고 있어야 하는 데이터 스트림의 튜플 수를 줄여 메모리 사용량도 크게 감소시켰다. 그리고 기존의 방법을 연속 다중 조인 질의의 처리에 단순하게 적용했을 때와 제안 방법을 서비스율과 메모리 사용량 측면에서 비교 분석하였다. 마지막으로 실험을 통해, 제안 방법이 기존 방법에 비해 서비스율과 메모리 사용량 측면에서 모두 우수한 성능을 보이고 있음을 보였다.

본 연구에서는 릴레이션들이 모두 하나의 디스크 내에 저장되어 있다고 가정하고 있

으며, 릴레이션들이 다수의 컴퓨터에 분산되어 저장되어 있는 경우는 고려하고 있지 않다. 대규모의 릴레이션들이 분산되어 저장되어 있는 경우는 네트워크 전송 비용, 접근 비용 등이 추가로 고려되어야 한다. 추후로는 이렇게 다수의 컴퓨터에 분산된 릴레이션을 고려한 연속 조인 처리 기법을 연구할 예정이다.

References

[1] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J., "Processing sliding window multi-joins in continuous queries over data streams," In Proc. ACM SIGMOD-SIGACTSIGART Symposium on Principles of Database Systems (PODS), Madison, Wisconsin, USA, pp. 1-16, June 2002.

[2] Garcia-Molina, H., Ullman, J. D., and Widom, J., DATABASE SYSTEMS : The complete Book : International Edition, 2/E. pp. 718-745, 2009.

[3] Golab, L. and Ozsu, T., "Processing sliding window multijoins in continuous queries over data streams," In Proc. Int. Conf. on Very Large Databases (VLDB), Berlin, Germany, pp. 500-511, September 2003.

[4] Kang, J., Naughton, J. F., and Viglas, S., "Evaluating window joins over unbounded streams," In Proc. Int. Conf. on Data Engineering, Bangalore, India, pp. 341-352, March, 2003.

[5] Karakasidis, A. and Hellas, I., "ETL queues for active data warehousing," In Proc. Int. Workshop on Information Quality in Informational Systems (IQIS), pp. 28-39, 2005.

[6] Lee, Y. W., Lee, K. Y., and Kim, M. H., "Multiple Continuous Skyline Query Processing over Data Streams," The Journal of Society for e-Business Studies, Vol. 15, No. 4, pp. 165-180, November 2010.

[7] Naeem, M. A., Dobbie, G., and Weber, G., "Optimised X-HYBRIDJOIN for Near-Real-Time Data Warehousing," In Proc. 23rd Australasian Database Conference, pp. 21-30, 2012.

[8] Naeem, M. A., Dobbie, G., and Weber, G., "X-HYBRIDJOIN for Near-Real-Time Data Warehousing," In Proc. 28th British National Conference on Databases, pp. 33-47, 2011.

[9] Naeem, M. A., Dobbie, G., Weber, G., and Alam, S., "R-MESHJOIN for Near-real-time Data Warehousing," In Proc. the ACM 13th International Workshop on Data Warehousing and OLAP, pp. 53-60, 2010.

[10] Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., and Frantzell, N., "Meshing Streaming Updates with Persistent Data in an Active Data Warehouse," IEEE Trans. on Knowl. And Data Eng., Vol. 20, No. 7, pp. 976-911, 2008.

[11] Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, and A., Frantzell, N., "Sup-

porting Streaming Updates in an Active Data Warehouse," In Proc. IEEE 23rd International Conference on Data Engineering, Istanbul, Turkey, pp. 476-485, 2007.

[12] Viglas, S. D., Naughton, J. F., and Burger, J., "Maximizing the output rate of multi-way join queries over streaming information sources," In Proc. Int. Conf. on Very Large Databases (VLDB), Berlin, Germany, pp. 285-296, September, 2003.

[13] White, C., "Intelligent business strategies: Real-time data warehousing heats up," DM Review, 2002.

저 자 소 개

김해리 (E-mail : haerik11@sookmyung.ac.kr)
2010년 숙명여자대학교 컴퓨터과학과 (학사)
2013년 숙명여자대학교 컴퓨터과학부 (석사)
관심분야 데이터 스트림 질의 처리

이기용 (E-mail : kiyonglee@sookmyung.ac.kr)
1998년 KAIST 전산학과 (학사)
2000년 KAIST 전산학과 (석사)
2006년 KAIST 전자전산학과 전산학전공 (박사)
2006년~2008년 삼성전자 기술총괄 소프트웨어연구소 책임연구원
2008년~2010년 KAIST 전산학전공 연구 조교수
2010년~현재 숙명여자대학교 컴퓨터과학과 조교수
관심분야 데이터베이스, 데이터 스트림, 질의 처리, 빅데이터