데이터 스트림 윈도우 질의를 위한 다중의 연속 MJoin 연산자 공유 처리

이헌주○ 박 석
서강대학교 컴퓨터학과
{hunz○, spark}@sogang.ac.kr

Sharing Multiple Continuous MJoins for Window Queries over Data Streams

Hunjoo Lee○ Seog Park
Dept. of Computer Science, Sogang University

요 약

데이터 스트림 관리 시스템에서 조인 연산자는 질의가 내포하는 여러 연산자들 가운데 상대적인 계산 비용이 높은 연산자로, 센서 네트워크와 같이 한정적 정보들이 개별적으로 입력되는 환경에서는 필연적으로 요구된다. 데이터 스트림은 잠재적으로 무한한 크기를 가지므로 조인 연산자는 슬라이딩 윈도우 제약사항을 가져야 하며, 종합적인 결과를 얻기 위해 조인 연산자가 여러 입력을 취할 수 있어야 한다. 이를 가능하게 하는 것이 바로 슬라이딩 윈도우를 가지는 MJoin 연산자이다. 본 논문에서는 이러한 여러 MJoin 연산자가 시스템에 등록되어 있는 환경을 가정하고, 슬라이딩 윈도우 제약사항과 MJoin의 특성을 반영하여 전역적으로 공유된 질의 실행 계획 수립 및 처리에 관한 문제를 다룬다. 이러한 다중 MJoin에 대한 전역 공유 질의 실행 계획 수립 문제가 NP-Hard임을 증명하고, 근사화 접근 방법을 제안한다. 또한 전역적으로 공유된 질의 실행 계획을 올바르게 수행할 수 있는 처리 기법을 제안한다. 이러한 연구의 노력은 데이터 스트림 환경에서 효율적인 다중 질의 최적화 및 처리기법의 기초 연구로 활용될 수 있다.

1. 서 론

최근의 데이터 연구는 잠재적으로 무한하고 연속적인 입력 스트림에 초점을 맞추기 시작했다. 이는 데이터 스트림(data stream)으로 언급되는데, 실시간으로 연속적이고 정렬된 일련의 아이템으로 정의할 수 있다. 이러한 환경에서는 전통적인 질의 처리에서 사용되었던 가정들이 더 이상 유효하지 않다[1]. 데이터 스트림의 가장 대표적인 환경은 센서 네트워크[2]이다. 센서 네트워크에서는 수 많은 센서가 일정 시간 간격으로 정보를 수집하여 데이터를 중앙 처리 서버로 전송한다. 이때, 하나의 센서가 수집하는 데이터는 그 처리 능력으로 인해 정보의 종류가 한정적이다[3]. 따라서 종합적인 정보를 얻고자 할 때, 특정 시간이나 위치를 기반으로 조인(join)연산을 수행하여 그 결과를 얻어야 한다. 오랜 기간 데이터베이스 연구분야에서는 여러 조인 연산 기법들이 제안되었다. 해시 테이블(hash table) 기반 조인 연산자[4-6], 윈도우(window) 기반 조인 연산자[7, 8], 해시 테이블-윈도우(hash table-window) 기반 조인 연산자[9]는 그러한 노력의 결과이다. 이 중 해시 테이블-윈도우 기반 조인 연산자는 한정된 메모리 내에서 작업이 가능하고, 빠른 매치 속도를 가진다는 점에서 데이터 스트림 환경에 가장 잘 맞는 처리 방법이다.

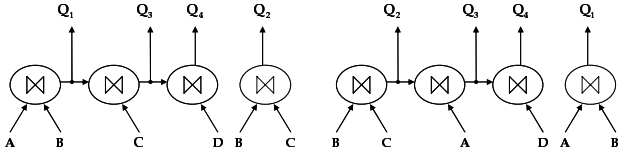
MJoin[9]은 각 입력에 대해 해시 테이블을 구성하며 새로운 튜플이 들어오면 해당하는 해시 테이블에 그 튜플을 삽입하고, 다른 해시 테이블을 조사하여 매치를 수행한다. 낮은 선택비율(selectivity)을 가지는 해시 테이블에서 높은 선택비율을 가지는 해시 테이블로 조사를 진행하여 매치 순서가 최적화된다. MJoin 연산자는 여러 입력을 취하며 최적화된 매치 순서를 가진다는 점에서 데이터 스트림 환경에 적합하다고 평가할 수 있다. 따라서 본 논문에서는 조인 연산 시 MJoin과 윈도우가 합쳐진 윈도우 기반 MJoin 연산자에 초점을 맞춘다.

본 논문은 여러 MJoin 연산자가 시스템에 등록되어 있을 때 각 연산자들간의 포함관계를 파악한 후, 전역 공유 질의 실행 계획(global shared query execution plan)을 수립하고, 수립된 질의 실행 계획을 올바르게 처리하는 목적을 가진다.

본 논문의 구성은 다음과 같다. 우선 2장에서는 본 연구에서 다루고자 하는 문제점을 정의한다. 3장에서는 본 연구와 관련된 타 연구들을 살펴보며, 4장에서는 2장에서 정의한 문제점들을 해결하기 위해 본 연구에서 제안하는 세 가지 기법을 소개하고, 5장에서는 실험을 통해 제안하는 기법의 처리 성능과 효율성을 분석한다. 마지막으로 6장에서는 본 연구의 결론 및 추후 연구 과제를 기술한다.

2. 문제 정의

2.1. 다중 MJoin에 대한 전역 공유 질의 실행 계획 수립 문제



[그림 1] 전역 질의 실행 계획들

[그림 1]에서와 같이 여러 질의가 MJoin으로 처리될 경우, 이 질의들로부터 생성될 수 있는 전역 공유 질의 실행 계획은 복수개가 될 수 있다. 다중 MJoin 연산자에 대한 전역 공유 질의 실행 계획 수립 문제는 여러 질의들로부터 생성되는 전역 공유 질의 실행 계획들 중 최소의 처리 비용을 가지는 하나를 선택하는 문제이다. 이를 다음과 같이 정의할 수 있다.

본 연구는 한국과학재단 특정기초연구(R01-2006-000-10609-0) 지원으로 수행되었음.

n개의 MJoin 연산자가 주어졌을 때, 각 MJoin M_i ($1 \le i \le n$) 는 가능한 질의 실행 계획 집합 $p_i=\{p_{i1},p_{i2},\ldots,p_{ik}\}$를 가진다. 각 M_i에 대해 p_i집합에서 하나의 질의 실행 계획을 선택하고, 여러 질의에 공유되는 부분은 병합하여 전역 질의 실행 계획을 수립한다. 이때, 가능한 모든 전역 질의 실행 계획들의 조합 중 전체 처리 비용이 최소가 되는 전역 질의 실행 계획 GP를 선택하는 것이 다중 MJoin에 대한 전역 공유 질의 실행 계획 수립 문제이다.

Sellis[13]는 일련의 질의들이 주어졌을 때, 여러 질의들에 대한 최적의 전역 질의 실행 계획을 수립하는 문제를 다루었는데, 그러한 문제가 NP-Hard임을 증명하였다. 먼저, 전통적인 다중 질의의 최적화 문제는 다음과 같다.

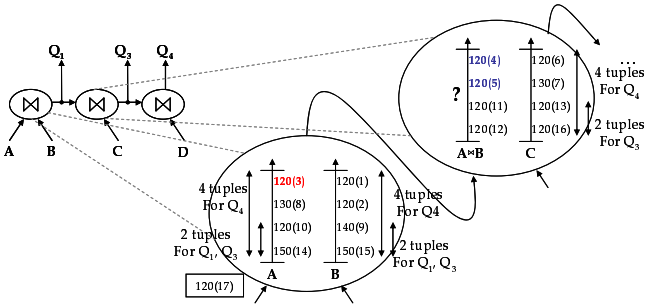
질의 Q_i ($1 \le i \le n$)에 대한 가능한 질의 실행 계획의 집합이 $p_i=\{p_{i1},p_{i2},\ldots,p_{iki}\}$라 하고, 모든 질의에 대한 n개의 질의 실행 계획 집합 $p_1,p_2,\ldots,p_n$이 주어졌을 때, 각 p_i에서 하나씩의 질의 실행 계획을 선택함으로써 최소 비용을 가지는 전역 질의 실행 계획 GP를 찾는 것이다. Sellis[10]는 이 문제를 3SAT 문제 [11]로부터의 변형(reduce)을 통해 NP-Hard임을 증명하였다. 다중 질의의 최적화 문제가 다중 MJoin에 대한 전역 공유 질의 실행 계획 수립 문제로 쉽게 변형될 수 있음은 다음과 같다.

정리 2.1 다중 질의의 최적화 문제 ∝ 다중 MJoin의 전역 공유 질의 실행 계획 수립 문제

증명 (1) 다중 질의의 최적화 문제에서의 최소 비용을 가지는 전역 질의 실행 계획은 다중 MJoin 연산자를 쪼갠 후 공유시켜 최소 비용을 가지는 전역 공유 질의 실행 계획과 같은 개념이다. (2) 다중 질의의 최적화 문제에서의 각 질의 Q_i는 각 MJoin 연산자 M_i와 같은 개념이다. (3) 각 질의 Q_i가 가질 수 있는 가능한 모든 질의 실행 계획 집합 p_i는 MJoin이 쪼개질 질 수 있는 가능한 모든 형태의 집합과 같다. 따라서, (1), (2), (3)에 의해 다중 질의의 최적화 문제는 어떠한 예에서도 본 연구에서 다루는 다중 MJoin에 대한 최적의 전역 공유 질의 실행 계획 문제의 예를 얻을 수 있다. 결국, 다중 질의 최적화 문제를 해결할 수 있으면 다중 MJoin에 대한 최적의 전역 공유 질의 실행 계획을 수립할 수 있으며 그 역도 성립한다. 다중 MJoin에 대한 최적의 전역 공유 질의 실행 계획을 수립하기 위해서는 근사화된 전략이 사용되어야 하며, 그러한 근사화 전략이 데이터 스트림 환경을 고려하고 적은 계산 비용만을 필요로 해야 한다.

2.2. 조인 연산 결과에 대한 윈도우 갱신 및 라우팅 문제

여러 MJoin을 공유하여 처리할 때에는 윈도우 갱신 시 문제가 발생 할 수 있다. [그림 2]에서 A⋈B와 C에 대해 조인을 수행하는 연산자에서 A⋈B의 튜플들은 입력된 순서대로 정렬되어 있지 않으므로 입력 A의 어떤 튜플이 제거가 될 경우, 상위 조인 연산자에서 A⋈B의 튜플들 중 제거해야 할 튜플을 빨리 찾을 수 없다. 마찬가지로, 각 질의에서 정의된 윈도우 크기로 조인된 결과를 적절한 질의 결과로 분기시켜 주어야 한다.



[그림 2] 전역 공유 질의 실행 계획에서 윈도우 갱신, 라우팅

3. 관련연구

3.1. 전역 공유 질의 실행 계획 수립에 관한 연구

3.1.1. 전통적인 다중 질의 최적화 기법

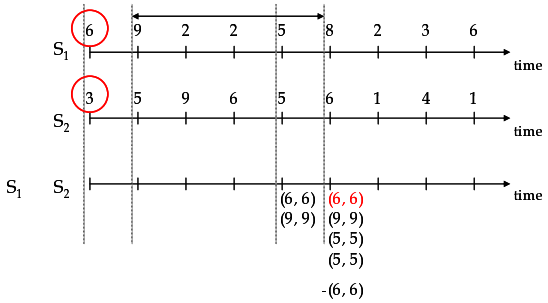
기존의 관계 형 데이터베이스(relational database)와 추론 데이터베이스 (deductive database)에서 여러 질의들을 처리해야 하는 요구가 증가하면서 다중 질의의 최적화 기법들 [10, 12]이 제안되었다. 이러한 다중 질의 최적화 기법 문제는 이미 2장에서 NP-Hard임을 설명하였다. 오랜 기간 이 문제를 근사화(approximation)하여 해결하기 위해 많은 기법들이 제안되었는데, 그 중 가장 대표적인 기법으로 A* greedy 기법[12]을 들 수 있다. A* greedy 기법은 매우 큰 탐색 공간에 대한 근사화 접근 방법인 A* search 알고리즘을 이용한 탐색 방법으로, 최적의 해에 근접하는 결과를 얻을 수 있는 효율적인 방법으로, 초기 상태(state)에서 마지막 상태까지 가장 적은 비용을 찾기 위해 상태 공간을 탐색한다. 따라서 이 기법은 최악의 경우 질의 당 모든 가능한 질의 실행 계획들의 수에 대해 지수 승에 비례하는 시간이 요구될 수 있으므로 공유 질의 실행 계획을 빨리 수립해야 하는 데이터 스트림 환경에는 부적합하다고 할 수 있다.

3.1.2. 그 밖의 최적화 기법

Dynamic Regrouping[14]은 질의가 계속 추가되는 환경에서 점진적인 질의 최적화를 수행하지만, 최초에 여러 질의가 한번에 주어지면 모든 가능한 질의 실행 계획을 살펴보아야 하며, 슬라이딩 윈도우 제약 사항을 고려하고 있지 않다. [15]에서는 공유되는 부분을 최대로 하는 전역 공유 질의 실행 계획 수립 기법을 제안하였으나 본래 연산자를 너무 많이 쪼개어 기존의 최적 매치 순서(probing sequence)를 많이 위배하게 되며, 스케줄러에 의한 문맥 교환(context switch)이 증가하여 성능이 하락할 수 있는 문제가 있다.

3.2. 조인 연산 결과의 윈도우 갱신에 관한 연구

[16, 17]에서는 조인 연산 결과에 대한 윈도우 갱신을 위해 Negative Tuple기법을 제안하였다. [그림 3]에서 튜플 6, 3이 윈도우를 벗어나게 되어 제거되어야 한다면, 6, 3은 다시 조인 연산자의 입력으로 들어가 튜플 6에 대한 Negative Tuple로 –(6, 6)을 생성된다. Negative Tuple인 –(6, 6)과 동일한 값을 가지는 (6, 6)튜플이 값 기반 탐색(value-based Search)을 통해 제거된다.



[그림 3] Negative Tuple 기법

3.3. 공유 조인 연산자를 위한 라우팅 기법에 관한 연구

본 연구에서는 라우팅을 위해 Dead Vector 기법[18]을 사용할 것임을 미리 밝혀둔다. Dead Vector 기법은 본래 조인 연산과 선택(selection) 연산의 공유 처리를 위해 제안된 기법으로 선택 연산을 수행할 때 튜플을 제거시키지 않고, 튜플의 Dead Vector에 매치 정보를 기록해 둔 채 조인 연산을 수행한다. 생성되는 결과 튜플에는 본래의 Dead Vector를 병합하여 그 정보로 라우팅을 수행한다.

4. 다중 MJoin의 공유 처리

4.1. 최적의 전역 공유 질의 실행 계획 수립 근사화 기법

다중 MJoin 연산자에 대한 최적의 전역 공유 질의 실행 계획을 수립하는 문제는 NP-Hard이므로 데이터 스트림에서 이 문제를 다룰 때 빠르게 최적해에 근접하는 근사화 접근 방법이 필요하다. 근사화 기법은 다음과 같은 조건을 만족시켜야 한다.

첫째, 탐색 공간을 줄이고, 적은 탐색으로도 만족할만한 성능 향상을 이끌어내야 한다. 둘째, MJoin연산자의 특성이 충분히 반영되어야 한다. 셋째, 데이터 스트림 환경에서 한정된 메모리만을 사용하게 해주는 슬라이딩 윈도우의 제약사항을 반영해야 한다. 이 조건들을 충족시키기 위해 다음과 같은 관찰들을 해 볼 수 있다.

첫째, MJoin의 특성 상 하나의 질의를 기준으로 탐색을 해 나간다면 탐색 공간을 매우 줄일 수 있다. 둘째, 연산자를 너무 많이 쪼갤 경우 오히려 성능 하락을 초래할 수 있다. 연산자 수가 증가하게 되면 문맥 교환이 증가하고 최적화된 매치 순서(probing sequence)가 너무 많이 위배된다. 셋째, 처리 비용이 높은 연산자를 많이 공유할수록 성능 향상 폭이 증가한다. 비용이 높은 연산자를 따로 처리하는 것보다 한번에 처리할 때 더 많은 이득을 볼 수 있다. 본 연구에서는 조인 연산자의 비용 모델(cost model)을 다음과 같이 정의한다.

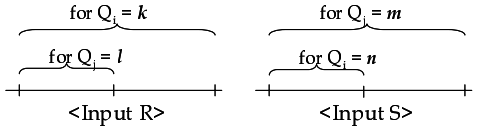
정의 4.1 어떤 MJoin이 n개의 입력 $R=\{R_1,R_2,\ldots,R_n\}$을 사용하고 각 R_i의 입력 속도와 윈도우 크기를 각각 r_i, W_i라 하자. f를 조인 연산에 대한 선택 비율 요소(Selectivity Factor)라 하면, 다음과 같이 비용 모델을 정의 할 수 있다.

$$\prod_{\substack{all\\ selectivities}} f \cdot \sum_{k=1}^{n} \left(r_k \cdot \prod_{\substack{i=1\\ i\neq k}}^{n} W_i \right)$$

또한, 각 조인 연산 결과에 대한 윈도우 크기를 다음과 같이 예측하여 상위 연산자에 대한 비용을 측정할 수 있다.

$$\prod_{\substack{all\\ selectivities}} f \cdot \prod_{i=1}^{n} W_i$$

마지막으로, 슬라이딩 윈도우 제약사항을 고려해야 한다. 조인 연산자들은 각 질의에서 정의된 윈도우가 다르더라도 공유되어 처리될 수 있다. 연산자가 공유될 때 가장 큰 윈도우 크기를 포함하게 되는데, 이 때문에 공유 시 오히려 더 많은 튜플을 저장하게 되는 가능성을 지닌다.



[그림 4] 두 입력에 대한 서로 다른 윈도우

만일 질의 Q_i와 Q_j가 입력 R과 S에 대한 조인을 수행하고, Q_i는 R과 S에 대해 각각 k와 n만큼의 윈도우를 사용하며, Q_j는 R과 S에 대해 각각 l과 m만큼의 윈도우를 사용한다고 하자. 이 두 질의가 공유되지 않는다면, 질의 Q_i는 최대 k×n만큼의 튜플에 대한 연산을 수행하고, Q_j는 최대 l×m만큼의 튜플에 대한 연산을 수행하게 된다. 만일 이 두 질의가 공유되어 처리된다면, 처리하는 튜플의 최대 양은 k×m이 된다. 연산자를 공유하기 전에 다음과 같은 조건에 부합하는지를 평가해야 한다.

정의 4.2 (*공유 조건 식*) m개의 공유 가능한 질의 Q_i ($1\le i\le m$)가 있고, 공유되는 MJoin 연산자가 k개의 입력 스트림 R_j ($1\le j\le k$)를 가진다고 하자. 또한, 각 질의 Q_i는 각 입력 R_j에 대해 W_{ij}의 윈도우 크기를 사용하며, 각 R_j에 대해 모든 질의에서 정의한 윈도우 크기 중 가장 큰 윈도우 크기를 W_j*라 하자. 다음 조건을 만족시켜야 연산을 공유시켰을 때 처리하는 양이 감소한다.

$$\sum_{i=1}^{m} \prod_{j=1}^{k} W_{ij} \ge \prod_{j=1}^{k} W_j *$$

위 관찰내용을 토대로 다중 MJoin 연산자의 최적의 전역 질의 실행 계획을 근사화 하여 수립하는 경험론적 욕심쟁이(heuristic greedy) 알고리즘을 제안한다.

제안하는 기법은 매 단계마다 하나의 MJoin을 선택한다. 각 단계에서는 가장 많이 공유되는 집합이 복수개가 선택된다면 그 중 가장 많은 처리 비용을 가지는 하나의 집합을 선택한다. 선택된 집합이 정의 4.2의 공유 조건 식에 만족하지 않는다면, 선택 집합을 제외하고 다음 단계로 넘어간다. 선택된 집합이 공유 조건 식에 부합하면, 그 집합을 포함하여 모든 주어진 집합에서 선택된 집합의 원소를 포함하는 부분을 찾아 해당 부분을 선택된 집합으로 치환한다. 마지막으로, 하나의 원소만을 가지는 집합을 다음 단계에서 제외한다. 이는 하나의 원소를 가지는 집합은 질의 실행 계획이 완성되었음을 의미하기 때문이다. 이 작업을 모든 질의가 제외될 때까지 반복한다.

제안하는 기법은 매 단계에서 하나의 원소만을 가지게 되는 집합이 제외되므로 매 단계에서는 적어도 하나의 집합이

제외된다. 따라서 [알고리즘 1]이 반환하는 연산자 수는 본래의 연산자 수보다 같거나 작게 된다.

```
Input : a set of queries, QuerySet[]
Output : shared query plans, QuerySet[]
while (QueryCount > 0) {
    for (i := 0 to QueryCount) {
        if (Containing # of SelectedSet < Containg # of QuerySet[i])
SelectedSet := QuerySet[i]
        else if (Containing # of SelectedSet = Containg # of QuerySet[i]
and Cost of QuerySet[i] > Cost of SelectedSet)
SelectedSet := QuerySet[i]
    }
    if (SelectedSet satisfies CONDITION OF SHARING) {
        for (i := 0 to QueryCount) {
            Replace the elements of QuerySet[i] to common elements of
SelectedSet
        }
        for (i := 0 to QueryCount) {
            if (QuerySet[i] has only one element) {
                exclude QuerySet[i]
                QueryCount := QueryCount - 1
    } } }
    else { exclude SelectedSet }
}
return QuerySet
```

[알고리즘 1] 경험론적 욕심쟁이 전략을 이용한 다중 MJoin의 최적화 알고리즘

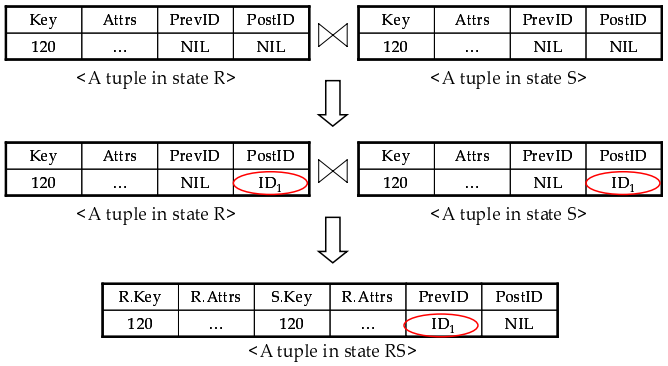
[알고리즘 1]의 입력으로 질의가 n개 주어지고 각 질의는 최대 m개의 입력을 가진다고 가정하자. 그렇다면 알고리즘의 매 단계마다 포함되는 질의 집합 수와 비용을 평가하여야 하는데 이는 각 질의가 최대 m개의 입력을 가지고 있으며 전체 질의 집합을 탐색해야 하므로 $O(nm^2)$ 의 시간이 소요된다. 또한 공통되는 원소들을 하나의 원소로 치환하는 작업 역시 선택된 집합을 포함하는 질의를 찾기 위해 모든 질의 집합을 검사해야 하며 각 질의가 최대 m개의 입력을 사용하므로 $O(nm^2)$의 시간이 소요된다. 공유 조건 식을 계산하기 위해서는 $O(nm+nm)=O(nm)$의 시간이 소요된다. 최악의 경우 [알고리즘 1]은 매 단계에서 하나의 질의만을 제외시키므로 n번 반복된다. 따라서 제안하는 기법의 총 시간 복잡도는 $O(n^2m^2+n^2m+n^2)$이 된다. 이는 최적화 시간을 다항시간(polynomial time)으로 줄이게 되었음을 말한다.

4.2. 조인 연산 결과를 위한 윈도우 갱신 및 라우팅 기법

4.2.1. 인덱스 할당

원시 스트림에서 윈도우 갱신으로 튜플이 제거되거나 라우팅 정보가 추가될 때마다 이와 관련된 모든 상위 튜플에도 영향을 미쳐야 한다. 본 연구에서는 이를 위해 인덱스 할당 기법을 제안한다. [그림 5]는 제안하는 인덱스 스킴을 나타낸 것으로 각 튜플이 PrevID와 PostID의 쌍을 추가적으로 가짐을 보인다. PrevID는 이 튜플이 어떠한 튜플들로부터 생성된 것인지를 알려주며, PostID는 이 튜플이 어떠한 튜플들을 생성한 것인지를 알려주게 된다. 튜플이 처음 시스템에 입력되면 PrevID와 PostID는 NULL값을 가진다. 두 튜플에 대한 조인이 수행되면, 해당하는 두 튜플의 PostID에 동일한 인덱스 값을 추가한다.

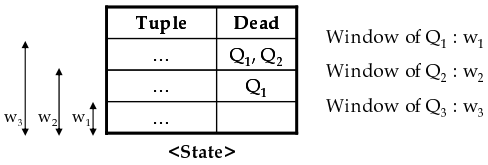
조인에 참여한 튜플이 매치되면서 추가된 인덱스 값과 동일한 값이 조인 연산 결과 튜플의 PrevID에 기입된다. 여기에서 주의해야 할 점은 PrevID는 하나의 값을 가지는 속성이며, PostID는 벡터(Vector)의 형태로 여러 인덱스 값을 유지한다는 것이다. 이는 하나의 튜플이 여러 조인 연산 결과 튜플을 생성하기 때문이다.



[그림 5] 조인 수행 시의 인덱스 할당

4.2.2. 라우팅을 위한 Dead Vector

본 논문에서는 3장에서 언급하였듯이 Dead Vector를 이용하여 라우팅을 위한 정보를 사전에 튜플 내에 기입하는 방법을 제안한다.



[그림 6] Dead Vector의 예

[그림 6]은 서로 다른 윈도우 크기를 사용하는 3개의 질의가 하나의 연산자로 공유되어 치리 되었을 때 Dead Vector의 사용 예를 나타낸 것이다. 질의 Q_1, Q_2, Q_3이 각각 w_1, w_2, w_3의 윈도우를 정의하였을 때, 공유된 조인 연산자는 가장 큰 윈도우 크기인 w_3만큼의 튜플들을 모두 조인 스테이트에 저장한다. 처음에 입력된 튜플은 모든 윈도우 내에 모두 포함되어 있으므로, Dead Vector에 아무런 값도 기입되지 않는다. 만약 다른 튜플이 입력되면 기존의 튜플은 w_1(1개의 튜플)을 벗어나게 되어 Dead Vector에는 Q_1을 기입하는데, 이것은 해당 튜플이 Q_1에 만족하지 않는 다는 것을 나타내는 정보이다. 만일 입력 스트림 L과 R이 조인된다면, 스테이트 L과 스테이트 R에서 키 값이 120인 튜플들이 조인 연산 결과 튜플을 생성하게 된다. 조인 연산 결과 튜플이 생성될 때에 조인에 참여하는 튜플이 가지고 있는 두 Dead Vector를 합치게 된다.

4.2.3. Purging Tuple과 Dead Tuple

Key	Attrs	PrevID	PostID
120	...	NIL	ID_1, ID_2, ID_3

(a) 윈도우 갱신에 의해 제거되는 튜플

Key	Attrs	PrevID	PostID
120	*	NIL	ID_1, ID_2, ID_3

(b) Purging tuple

[그림 7] Purging Tuple 생성의 예

원시 스트림의 튜플에서 어떤 튜플이 제거되거나 튜플에 새로운 Dead 값이 추가될 때마다 상위 조인 연산자의 연산 결과 튜플에도 영향을 끼쳐야 한다. 본 논문에서는 이를 위해 Purging Tuple과 Dead Tuple을 제안한다. 이들은 Negative Tuple

기법을 변형한 형태로, Negative Tuple과는 인덱스 기반 탐색을 수행한다는 점과 이들을 생성하기 위해 다시 조인 연산을 거치지 않는다는 점이 다르다.

인덱스 정보를 이용하여 윈도우 갱신 시 튜플이 제거되었을 때 이와 연관된 튜플을 삭제해 주어야 한다. 이를 위해 생성되는 튜플이 Purging Tuple이다. Purging Tuple은 인덱스 정보와 특별한 플래그(Flag) 를 가지며 실제 조인 연산에는 참여하지 않고 튜플을 삭제하는 데에만 사용된다. [그림 7b]는 [그림 7a]의 튜플이 제거되면서 생성되는 Purging Tuple로, 이 튜플이 Purging Tuple임을 알려주는 플래그가 Attrs속성에 기입되어 있으며, 제거되는 튜플과 동일한 PostID 값을 가진다. 이렇게 생성된 Purging Tuple은 상위 조인 연산자로 이동한다.

Key	Attrs	PrevID	PostID	Dead
120	...	NIL	ID_1, ID_2	Q_1

(a) 새로운 Dead Vector 값이 추가된 튜플

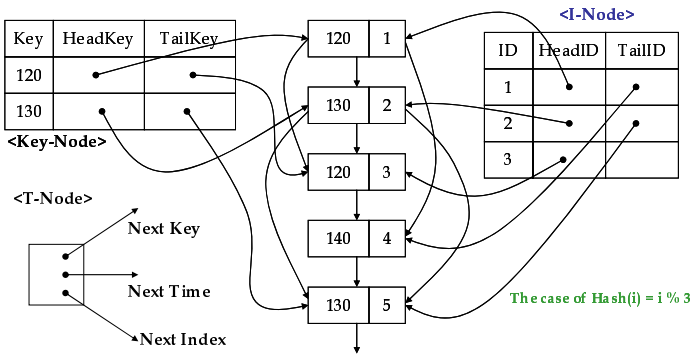
Key	Attrs	PrevID	PostID	Dead
120	**	NIL	ID_1, ID_2	**Q_1**

(b) Dead Tuple

[그림 8] Dead Tuple 생성의 예

어떠한 원시 스트림 튜플의 Dead Vector에 새로운 값이 추가되면, 이 튜플이 생성한 조인 연산 결과 튜플의 Dead Vector에도 동일한 값을 추가하여야 한다. 본 연구에서는 Dead Vector에 값을 추가할 수 있는 정보를 운반하는 튜플로 Dead Tuple을 제안한다. Dead Tuple이 상위 조인 연산자에 전달되면 Dead Tuple의 PostID값을 얻고, 인덱스 해시 테이블을 이용하여 조인 스테이트의 같은 PrevID를 가진 튜플들을 탐색하며, 매치되는 튜플에 Dead Tuple이 가진 Dead Vector를 결합시킨다. [그림 8]은 이러한 Dead Tuple의 예를 보여준다. Dead Tuple은 Purging Tuple과 다른 플래그를 가지는데, 본 연구에서는 이를 이중 Asterrisk(**)로 표시한다.

4.2.4. 인덱스 해시 테이블



[그림 9] 조인 연산자의 자료구조

Purging Tuple과 Dead Tuple이 해당 튜플을 제거하거나 새로운 Dead Vector 값을 추가하기 위해서는 인덱스 기반 탐색을 수행한다. 즉, 어떤 튜플의 PrevID값이 Purging Tuple 및 Dead Tuple이 가지고 있던 PostID에 포함된다면 그 튜플은 제거되거나 새로운 Dead Vector값이 추가된다. 본 연구에서는 인덱스 기반의 탐색을 보다 빠르게 수행하도록 인덱스 해시 테이블(Index Hash Table)을 조인 연산자 내에 구성한다. 인덱스 해시 테이블은 해당 조인 연산자내의 튜플들이 가지고 있는 PrevID를 기반으로 구성된다.

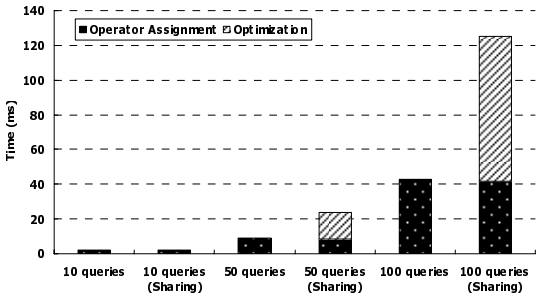
[그림 9]에서 <I-Node>는 인덱스 해시 테이블의 헤더(Header)정보로 각 PrevID의 값이 가장 처음으로 발생되는 튜플과 마지막에 나타나는 튜플에 대한 포인터(Pointer)를 유지한다. 각 튜플이 저장되는 자료구조는 <T-Node>로 표현되는데, 이들은 같은 PrevID를 가지는 다음 튜플에 대한 포인터를 유지함으로써 인덱스 해시 테이블의 계속적인 탐색을 가능하게 한다.

5. 성능평가

5.1. 실험 환경 설정

본 실험에서는 20개의 스트림을 가정하였으며 각 스트림의 입력 속도를 초당 300개 튜플로 정의하였다. 모든 튜플에서 키 속성 값은 0부터 1000까지의 정수 값을 가지도록 하였다. 이 키 속성 값은 모든 튜플에 걸쳐 같은 빈도수로 발생하도록 하였는데, 발생 순서는 무작위(Random)로 하여 슬라이딩 윈도우 내에 있는 키 속성 값들이 매번 바뀌므로 조인 연산자의 선택비율도 계속적으로 변화한다. 각 질의는 최대 20개의 서로 다른 입력 스트림을 사용할 수 있다. 실험의 정교함을 위해 여러 질의에 걸쳐 나타나는 입력 스트림들의 비대칭도(skewness)를 지프 분포(Zipf distribution) [19]를 이용하여 반영하였다. 본 실험에서 이를 SO라 명명한다. 또한, 각 질의에서 각각의 입력 스트림에 대해 500개의 튜플, 1000개의 튜플, 1500개의 튜플 중 하나를 무작위로 선택하여 윈도우로 사용하였으며, 각 윈도우는 균등 분포로 발생하도록 하였다. 이는 공유 조건 식에 부합하지 않는 경우가 발생할 수 있도록 하기 위함이다.

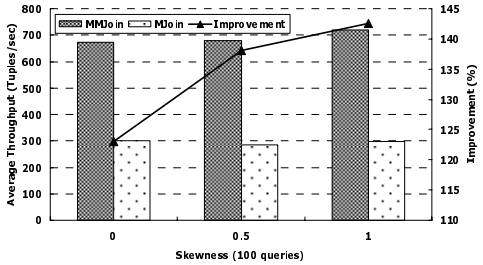
5.2. 실험1: 전역 공유 질의 실행 계획 수립 시간 비교



[그림 10] 질의 수 변화에 따른 수립 시간 비교 (SO=0.5)

[그림 10]은 각 질의에서 나타나는 입력 스트림의 비대칭도가 0.5일 때, 독립적으로 수행되는 MJoin과 제안하는 기법의 최적화 시간 및 연산자 할당 시간을 보여준다. [알고리즘 1]을 통해 연산자의 수가 줄어들기 때문에 본래의 연산자 할당 시간보다 약간 적은 시간이 소요되지만, 제안하는 기법에서는 질의 수가 증가할수록 별도의 최적화를 위한 연산이 증가하여 더 많은 시간이 요구된다. 실제 데이터 스트림 관리 시스템 엔진으로 내장된다면 더 빠른 시간 내에 최적화를 수행할 수 있을 것이라 예상된다.

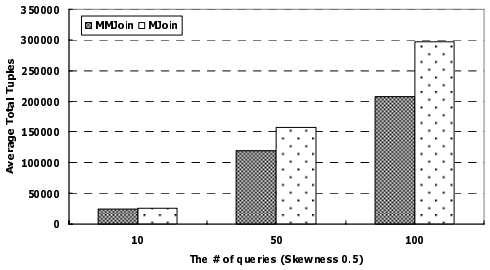
5.3. 실험2: MJoin과 제안하는 기법의 Throughput



[그림 11] 평균 처리량 비교 (Q=100)

[그림 11]은 질의의 수를 100개로 고정하고 입력 비대칭도를 변화시켰을 때의 평균 처리량을 보여준다. 입력 비대칭도가 증가할수록 여러 질의들 가운데에 공통되는 부분도 증가하여 더 많이 공유되어 제안하는 기법의 상대적인 평균 처리량도 증가함을 볼 수 있다. 질의의 수와 입력 스트림의 비대칭도가 증가하면 각 질의들이 포함관계를 가지는 경우가 증가하므로 제안하는 기법에서 공유되는 연산자의 수도 증가하게 된다. 일정 수준 이상 SO에 이르면 처리향상률의 큰 증가를 보이지 않는데 이는 일정 수준의 질의 수가 주어지면 이미 연산자가 충분히 공유될 수 있는 환경을 갖추기 때문이다. 질의 수 변화에 따른 처리량 변화는 공간의 제약으로 생략한다.

5.3. 실험3: MJoin과 제안하는 기법의 튜플 사용량



[그림 12] 질의 수에 따른 튜플량 비교 (SO=0.5)

[그림 12]은 SO를 0.5로 고정시키고 질의의 수를 변화시켰을 때 평균적인 튜플 사용량을 보여준다. 질의의 수가 증가할수록 제안하는 기법이 사용하는 상대적인 튜플 사용량이 점차 감소하는 것을 알 수 있다. 이는 질의 수가 증가하면 연산자가 더 많이 공유되기 때문이다.

6. 결론 및 추후연구

본 논문은 기존의 데이터 스트림 연구 분야에서 제안된 효율적인 조인 연산자를 이용하여 다중 질의로 그 영역을 확장하였고 효율적인 처리 기법을 제안하였다. 제안하는 기법들의 특징은 다음과 같다.

첫째, 데이터 스트림에서 전역 공유 질의 실행 계획의 수립 기법 시간을 대폭 줄였다. 둘째, 공유 질의 실행 계획 하에서 정확한 윈도우 갱신 기법을 제안하였다. 셋째, 공유 질의 실행 계획 하에서 정확한 라우팅 기법을 제안하였다.

유비쿼터스 환경과 같이 많은 사용자가 접근하는 시스템에서는 데이터 스트림에 대한 보다 빠른 처리가 요구되는데, 제안하는 기법이 이러한 환경에 보다 빠른 처리를 가능케 해줄 것으로 예상된다.

본 논문에서의 Purging/Dead Tuple 기법을 하나의 튜플로 통합하여 생성되는 튜플의 수를 줄이는 것과 동적인 환경에서 전역 공유 질의 실행 계획을 매우 적은 비용으로 수립할 수 있는 기법을 추후 연구과제로 남기고자 한다.

참고문헌

[1] B. Babcock, et al., "*Models and Issues in Data Stream Systems*", In PODS'02, pp.1-16, 2002.

[2] P. Bonnet, et al., "*Towards Sensor Database Systems*". In MDM'01, pp.3-14, 2001.

[3] Sven Schmidt, Marc Fiedler, Wolfgang Lehner, "*Source-aware join strategies of sensor data streams*", In SSDBM'05, pp.123-132, 2005.

[4] A. N. Wilschut and P. M. G. Apers. "*Pipelining in query execution*." In Conference on Database, Parallel Architectures and their Applications, p.562, 1991.

[5] T. Urhan and M. J. Franklin. "*XJoin: A reactively-scheduled pipelined join operator*." IEEE Data Engineering Bulletin, Volume 23, No. 2, pp.27-33, 2000.

[6] S. D. Viglas, J. F. Naughton, and Josef Burger. "*Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources*." In VLDB'03, pp. 285-296, 2003.

[7] L. Golab and M. T. Ozau. "*Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams*." In VLDB'03, pp.500-511, 2003.

[8] J. Kang, J. F. Naughton, and S. D. Viglas. "*Evaluating Window Joins over unbounded Streams*." In ICDE03, pp.341-352, 2003.

[9] L. Ding and E. A. Rundensteiner. "*Evaluating Window Joins over Punctuated Streams*." In CIKM'04, pp.98-107, 2004.

[10] T. Sellis, S. Ghosh, "*On the Multiple-Query Optimization Problem*", IEEE Transactions on Knowledge and Data Engineering, Volume 2 Issue 2, pp.262-266, 1990.

[11] M. R. Garey and D. S. Johnson, "*Computers and Intractability*." San Francisco, CA: Freeman, 1979.

[12] Timos K. Sellis, "*Multiple-query optimization*," ACM Transactions on Database Systems, Volume 13, Issue 1, pp.23-52, 1988.

[14] Jianjun Chen, David J. DeWitt, "*Dynamic Re-grouping of Continuous Queries*", In VLDB'02, pp.430-441, 2002.

[15] Yousuke Watanabe, Hiroyuki Kitagawa, "*A Multiple Continuous Query Optimization Method Based on Query Execution Pattern Analysis*", DASFAA'04, LNCS 2973, pp.443-456, 2003.

[16] M. A. Hammad, et al.. "*Efficient Pipelined Execution of Sliding Window Queries over Data Streams*." In Technical Report CSD TR#02-010, June, 2004.

[17] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid, "*Exploiting predicate-window semantics over data streams*." ACM SIGMOD Record, Volume 35, Issue 1. March, pp.555-568, 2006.

[18] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, Garrett Jacobson, "*The Case for Precision Sharing*", In VLDB'04, pp.972–986, 2004.

[19] C. D. Manning and H. Schütze. "*Foundations of Statistical Natural Language Processing.*" The MIT Press, 1999.