데이타 스트림 상에서 다중 연속 복수 조인 질의 처리 최적화 기법

(MMJoin: An Optimization Technique for Multiple Continuous MJoins over Data Streams)

변 창 우 † 이 헌 주 †† 박 석 †††
(Changwoo Byun) (Hunzu Lee) (Seog Park)

요 약 센서 네트워크에 이용되는 데이타 스트림 관리 시스템에서는 한정적 정보들이 개별적으로 입력되기 때문에 종합적인 결과를 얻기 위해서는 상대적인 계산 비용이 높은 조인 연산자는 필연적으로 요구된다. 데이타 스트림은 잠재적으로 무한한 크기를 가지므로 조인 연산자는 슬라이딩 윈도우 제약사항을 가져야 함은 당연하다. 또한, 종합적인 결과를 얻기 위해 조인 연산자는 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 윈도우를 가지는 MJoin의 특성을 반영하여 전역적으로 공유된 질의 처리 기법인 MMJoin 기법을 제안한다. MMJoin 기법은 첫째, 전역적으로 공유된 질의 실행 계획 수립 문제, 조인 연산 결과에 대한 윈도우 갱신 문제 및 라우팅 문제로 나누어 다룬다. 이러한 연구의 노력은 데이타 스트림 환경에서 효율적인 다중 질의 최적화 및 처리 기법의 기초 연구로 활용될 수 있다.

키워드 : 데이타 스트림, 다중 조인, 공유 처리, 다중 질의 최적화

Abstract Join queries having heavy cost are necessary to Data Stream Management System in Sensor Network where plural short information is generated. It is reasonable that each join operator has a sliding-window constraint for preventing DISK I/O because the data stream represents the infinite size of data. In addition, the join operator should be able to take multiple inputs for overall results. It is possible for the MJoin operator with sliding-windows to do so. In this paper, we consider the data stream environment where multiple MJoin operators are registered and propose MMJoin which deals with issues of building and processing a globally shared query considering characteristics of the MJoin operator with sliding-windows. First, we propose a solution of building the global shared query execution plan. Second, we solve the problems of updating a window size and routing for a join result. Our study can be utilized as a fundamental research for an optimization technique for multiple continuous joins in the data stream environment.

Key words : Data stream, Multiple Join, Sharing, Multi-query Optimization

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

† 정 회 원 : 인하공업전문대학 컴퓨터시스템과 교수
cwbyun@inhatc.ac.kr
†† 학생회원 : 서강대학교 컴퓨터공학과
hunz@dblab.sogang.ac.kr
††† 종신회원 : 서강대학교 컴퓨터공학과 교수
spark@sogang.ac.kr

논문접수 : 2007년 9월 5일
심사완료 : 2007년 11월 30일

정보과학회논문지: 데이타베이스 제35권 제1호(2008.2)

1. 서 론

최근의 데이타 연구는 고정된 저장 데이타가 아닌, 잠재적으로 무한하고 연속적인 입력 스트림에 초점을 맞춘 데이타 스트림(data stream)의 특성을 고려한 관리 시스템 개발에 관한 연구 프로젝트가 진행되었다[1-4]. 데이타 스트림 관리 시스템의 대부분의 질의는 연속 질의(continuous query)의 형태로 시스템에 등록되고, 데이타는 그 크기가 잠재적으로 무한하므로 모두 저장되지 못하고, 한정된 메모리 상에서 매우 빠르게 처리되며 그 결과는 사용자나 응용에 다시 스트림의 형태로 전달된다.

데이타 스트림 형태의 정보가 발생하는 가장 대표적인 환경은 센서 네트워크[5]이다. 센서 네트워크에서는 수 많은 센서로부터 일정 시간 간격으로 정보를 수집하며, 이렇게 얻은 데이타를 중앙 처리 서버로 전송한다. 다양한 센서로부터 다양한 데이타가 수집되지만, 하나의 센서로부터 수집되는 데이타는 그 처리 능력으로 인해 정보의 종류가 한정적이다[6]. 따라서 종합적인 정보를 얻고자 할 때, 특정 시간이나 위치를 기반으로 조인(join) 연산을 수행하게 된다. 해시 테이블(hash table) 기반 조인 연산자[7-9], 윈도우(window) 기반 조인 연산자[10,11], 해시 테이블-윈도우(hash table-window) 기반 조인연산자[12]는 그러한 노력의 결과이다. 추가로, 여러 정보들을 조인한 결과가 보다 종합적인 내용을 포함한다는 관점에서 다중 입력을 취할 수 있는 MJoin 연산자가 제안되었다[9]. MJoin은 기존의 해시 테이블 기반 조인 연산자인 XJoin[8]과 처리 방식이 동일하지만 다중 입력을 취한다는 점에서 차이점을 보인다.

그림 1은 MJoin의 처리 절차를 나타낸 것인데, MJoin은 각 입력에 대해 해시 테이블을 구성하며 새로운 튜플이 들어오면 해당하는 해시 테이블에 그 튜플을 삽입하고, 다른 해시테이블을 조사하여 매치를 수행한다. MJoin 연산자는 다중 입력을 취하는 특성 때문에 해시 테이블을 조사하는 순서가 성능에 큰 영향을 끼칠 수 있다. 이는 낮은 선택비율(selectivity)을 가지는 해시 테이블에서 높은 선택비율을 가지는 해시 테이블로 조사를 진행하여 최적화시킬 수 있다.

본 연구에서는 조인연산 시 여러 입력을 취하는 MJoin 연산자가 이용되고 이것에 윈도우가 합쳐진 윈도우 기반 MJoin 연산자에 초점을 맞춘다.

여러 질의들이 각각하나의 MJoin연산자로 처리된다면 다소 비효율적인 측면이 있다. 예를 들어, 조인 질의가 시스템에 그림 2(a)와 같이 등록되어 있다고 가정하자. 각 질의들은 부분적으로 중복된 연산을 수행하게 된다. 그림 2(a)의 질의 Q_1은 Q_3, Q_4에 포함될 수 있으며, Q_2도 Q_3과 Q_4에 포함될 수 있다. 이 질의들을 독립적인 MJoin 연산자로 처리한다면, 한번만 처리하여도 될 연산을 두 번 이상 수행하게 된다. Q_1이나 Q_2를 처리한 결과는 Q_3과 Q_4에서 사용될 수 있으므로 Q_1이나 Q_2를

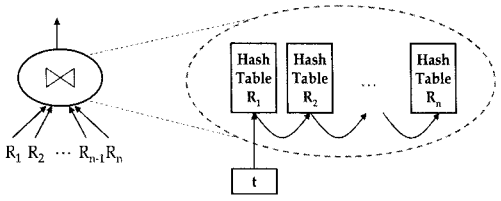
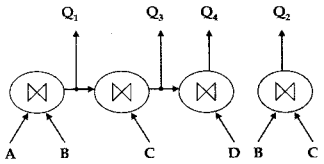


그림 1 MJoin 연산자의 예

Q_1=A[w_{11}] ⋈ B[w_{12}]
Q_2=B[w_{21}] ⋈ C[w_{22}]
Q_3=A[w_{31}] ⋈ B[w_{32}] ⋈ C[w_{33}]
Q_4=A[w_{41}] ⋈ B[w_{42}] ⋈ C[w_{43}] ⋈ D[w_{44}]

(a) 슬라이딩 윈도우 제약사항이 정의된 질의들



(b) 공유된 MJoin 연산자

그림 2 동기 부여 예제

처리한 결과를 Q_3과 Q_4의 입력으로 보내 처리한다면 중복 연산을 방지하여 질의 처리 속도를 향상시킬 수 있다. 그림 2(b)는 그림 2(a)의 질의들을 공유하여 처리할 수 있는 MJoin 연산자들을 나타낸 것이다.

본 연구는 그림 2에서와 같이 여러 MJoin 연산자가 시스템에 등록되어 있을 때 각 연산자들 간의 포함관계를 파악한 후, 전역 공유 질의 실행 계획(global shared query execution plan)을 수립하고, 수립된 질의 실행 계획을 올바르게 처리하는 목적을 가진다. 이와 같은 공유 처리 기법을 MMJoin (Multiple MJoin) 기법이라 명한다.

MMJoin 기법은 데이타 스트림 환경에서 슬라이딩 윈도우로 인한 튜플 사용량과 MJoin 연산자만이 가지는 다중 입력 지원 및 최적화된 평가 순서를 고려하여 다중MJoin 연산자에 대한 최적화를 이끌어낸다. 또한, 어떤 조인 연산자의 처리 결과가 다시 상위의 조인 연산자의 입력으로 사용될 수 있다. 이러한 환경에서 조인 연산 결과에 대한 윈도우 갱신 문제를 해결하는 인덱스 기반의 Purging Tuple을 제안한다. 마지막으로 조인 연산 결과 튜플의 라우팅 문제를 해결하기 위해Dead Vector 기법과 상위 조인 연산자에 새로 추가된 Dead 값을 전파시키는 Dead Tuple을 제안한다. Dead Tuple은 Purging Tuple과 같이 인덱스 기반 탐색을 하며 조인 연산 결과 튜플이 어떠한 윈도우에 만족되지 않는 것인가를 확인하는 데에 사용된다.

본 논문의 구성은 다음과 같다. 2장에서는 본 연구에서 다루고자 하는 문제점을 정의한다. 3장에서는 본 연구와 관련된 타 연구들을 살펴보며, 이러한 연구 내용들의 문제점을 분석하고, 본 연구 내용과 어떻게 구별되는지를 살펴본다. 4장에서는 2장에서 정의한 문제점들을 해결하기 위해 본 연구에서 제안하는 MMJoin 기법 내의 세 가지 하부 기법을 소개하며, 5장에서는 실험을 통해 MMJoin 처리 성능과 효율성을 분석한다. 마지막으

로 6장에서는 본 연구의 결론 및 추후 연구 과제를 기술한다.

2. 이슈 정의

2.1 MMJoin에 대한 전역 공유 질의 실행 계획 수립

MJoin 연산자는 다중 입력을 취하므로 다른 연산자의 일부가 될 수 있거나 부분적으로 공통되는 입력 스트림을 사용할 수도 있다.

그림 3에서와 같이 여러 질의들이 MJoin 연산자로 처리되는 경우, 이 질의들로부터 생성될 수 있는 전역 공유 질의 실행 계획은 복수개가 될 수 있다. 이와 같이 MMJoin 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. Sellis[13]는 데이타베이스 시스템에 일련의 질의들이 주어졌을 때, 여러 질의들에 대한 최적의 전역 질의 실행 계획을 수립하는 문제를 다루었는데, 그러한 문제가 NP-Hard임을 증명하였다. 이와 같은 전통적인 다중 질의의 최적화 문제로부터 MMJoin에 대한 전역 공유 질의 실행 계획 수립 문제가 NP-Hard임을 증명할 수 있다.

정의 2. MMJoin의 전역 공유 질의 실행 계획 수립 문제는 NP-Hard이다.

증명. n개의 MJoin 연산자가 주어졌을 때, 각 MJoin 연산자 M_i $(1 \le i \le n)$는 가능한 질의 실행 계획 집합 $p_i = \{P_{i1}, P_{i2}, ..., P_{ik_i}\}$을 가진다. (1) 우선 다중 질의의 최적화 문제에서의 최소 비용을 가지는 전역 질의 실행 계획은 MMJoin 연산자를 쪼갠 후 공유 시켜 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 수 있는 가능한 모든 형태의 집합과 같다.

(1), (2), (3)에 의해 다중 질의의 최적화 문제는 어떠한 예에서도 본 연구에서 다루는 MMJoin에 대한 최적의 전역 공유 질의 실행 계획 문제의 예를 얻을 수 있다. 결국, 다중 질의 최적화 문제를 해결할 수 있으면 MMJoin에 대한 최적의 전역 공유 질의 실행 계획을 수립할 수 있으며 그 역도 성립한다.

따라서, MMJoin에 대한 최적의 전역 공유 질의 실행 계획을 수립하기 위해서는 데이타 스트림 환경에 적합하도록 적은 계산 비용만을 필요로 하는 근사화된 전략이 사용되어야 한다. 또한 MJoin과 슬라이딩 윈도우의 특성이 충분히 반영된 공유 질의 실행 계획의 수립이 이루어져야만 한다. 이러한 문제를 해결하기 위한 알고리즘은 4장에서 제안한다.

2.2 조인 연산 결과에 대한 윈도우 갱신

MMJoin에 대한 전역 공유 질의 실행 계획이 수립된 이후에는 조인 트리(join tree)를 형성하기 때문에 윈도우를 갱신시켜 나아갈 때 문제가 발생 할 수 있다.

그림 4는 MMJoin에 대한 전역 공유 질의 실행 계획이 수립된 예를 나타낸 것이다. 위 그림에서 튜플 "120(1)"의 120은 조인 키 값을 나타낸 것이고, 괄호 안

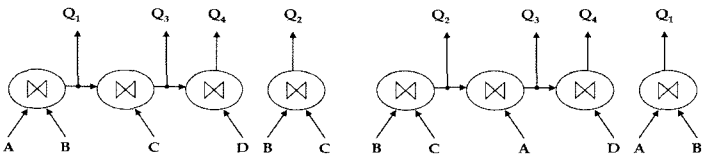


그림 3 그림 2(a)의 질의로부터 생성 가능한 전역 질의 실행 계획들

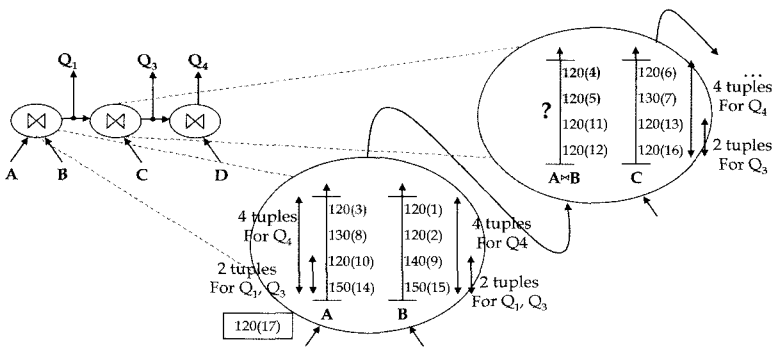


그림 4 전역 공유 질의 실행 계획에서의 윈도우 갱신 및 라우팅

의 숫자는 입력되는 순서를 나타낸 것이다. 입력 스트림 A와 B에 대해 조인 연산을 수행하는 연산자는 공유된 상태이며, 입력으로는 A와 B에 대해 정의된 윈도우 크기중 가장 큰 것을 취하게 된다. 만일 질의 Q_4가 A와 B에 대해 가장 큰 윈도우 크기(4 tuples)를 정의하였다면 이를 공유된 MJoin 연산자의 입력으로 취하게 된다. A와 B에 대해 공유된 조인 연산자는 원시 입력스트림만을 취하여 튜플들이 입력된 순서대로 연산자 내에 정렬되어 있기 때문에 윈도우갱신은 먼저 입력된 튜플을 우선적으로 제거함으로써 수행한다. 하지만 A⋈B와 C에 대해 조인을 수행하는 연산자에서 A⋈B의 튜플들은 하위 조인 연산자의 결과튜플들이므로 시스템에 입력된 순서대로 정렬되어 있지 않다. 만일, A와 B의 조인을 수행하는 연산자에서 입력 A의 어떤 튜플이 제거가 될 경우, 이와 관련된 상위 조인 연산자의 튜플도 제거가 되어야 하는데, 상위 조인 연산자에서 A⋈B의 튜플들 중 제거해야 할 튜플을 쉽게 찾을 수 없다. 결국, 상위 조인 연산자들의 윈도우 갱신을 위해 모든 튜플들에 대해 해당 튜플이 어떠한 튜플을 조인 연산 결과로 생성하였는지, 혹은 해당 튜플이 어떠한 튜플들의 조인 연산으로 생성되었는지를 파악할 수 있어야 한다.

2.3 조인 연산 결과에 대한 라우팅

각 질의에서의 윈도우 크기가 서로 다른 상황에서 MJoin 연산자가 공유될 때에는 각 질의가 정의한 윈도우 내의 튜플들에 대한 결과만을 내보내야 한다. 그림 4에서 A와 B에 대해 조인을 수행하는 경우에 연산자 내에 2가지의 윈도우를 포함하게 되고 총 3개의 질의에 대한 중간 결과를 내보내게 된다. 질의 Q_1과 Q_3이 2개의 튜플을 입력 A와 B에 대해 윈도우로 정의하고, 질의 Q_4가 4개의 튜플을 입력 A와 B에 대해 윈도우로 정의하였다면, 공유된 연산자의 모든 결과 튜플을 Q_1, Q_3, Q_4로 모두 내보낼 수 없다. 최근 튜플들이 2개의 튜플 수를 벗어난다면, Q_1, Q_3로 질의 결과를 내보낼 수 없고 Q_4로만 그 결과를 내보내야 한다. 원시 스트림인 A와 B에 대한 조인을 수행하는 연산자에서는 모든 튜플들이 입력된 순서대로 정렬되어 있기 때문에 타임스탬프(timestamp)나 입력 순서 번호를 판별하여 라우팅을 진행할 수 있다. 하지만 A⋈B와 C에 대해 조인을 수행하는 연산자에서 A⋈B에 대한 튜플들은 원시 스트림에 대한 라우팅과 같이 쉬운 전략을 수행할 수 없다.

다음 장에서는 본 장에서 정의한 문제들과 관련된 기존의 관련 연구들을 살펴보고 그 연구들이 본 연구에서 가정하는 환경에서 어떠한 문제점을 가지는지 분석한다.

3. 관련 연구

3.1 전역 공유 질의 실행 계획 수립에 관한 연구

기존의 관계형 데이타베이스(relational database)와 추론 데이타베이스 (deductive database)에서 대표적인 다중 질의의 최적화기법으로 A* greedy 기법[13]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 방법이다. 그러나, 최악의 경우 질의 당 모든 가능한 질의 실행 계획들의 수에 대해 지수 승에 비례하는 시간이 요구된다. 또한, 초기의 계산 비용이 너무 크므로 빠른 속도로 공유 질의 실행 계획을 수립해야 하는 데이타 스트림 환경에는 부적합하다고 할 수 있다.

Dynamic Regrouping[14]은 연속 질의에 대한 공유 질의 실행 계획 수립 기법의 하나로 질의가 동적으로 추가되는 환경에서 점진적인 질의 최적화를 수행한다. 이 기법은 크게 2단계를 거친다. 첫 번째 단계에서는 기존에 수립된 질의 실행 계획과 새로 입력된 질의의 실행 계획을 합쳐 가능한 포함관계를 파악한 후, 어떤 질의 실행 계획이 다른 질의 실행 계획에 포함된다면 링크로 연결시킨다. 두 번째 단계에서는 첫번째 단계에서 생성된 모든 링크들 가운데에 가장 상위에서부터 차례로 내려가면서 가장 비용이 작은 링크를 선택한다. 이렇게 선택된 링크를 확인함으로써 공유된 질의 실행 계획을 반환하게 된다. 이 기법은 최초에 여러 질의가 한번에 주어지면 결국 모든 가능한 질의 실행 계획을 모두 살펴보아야 하며, 슬라이딩 윈도우 제약 사항을 고려하고 있지 않기 때문에 본 연구에서 초점을 맞춘 슬라이딩 윈도우를 가진 MJoin의 특징이 반영될 수 없다.

[15]에서는 여러질의에 대해 모든 가능한 질의 실행계획이 주어지면 공유되는 부분을 최대로 하는 전역 공유 질의 실행 계획 수립 기법을 제안하였다. 하지만, 공유되는 부분을 최대로 하는 전역 공유 질의 실행 계획은 MJoin 연산자를 너무 많이 쪼개어 기존의 최적 매치 순서(probing sequence)를 너무 많이 위배하게 되며, 스케줄러에 의한 문맥 교환(context switch)이 증가하여 오히려 성능이 하락할 수 있는 문제가 있다.

3.2 조인 연산 결과의 윈도우갱신에 관한 연구

[16]에서는 조인 연산 결과에 대한 윈도우갱신을 지원하기 위해 Negative Tuple 기법을 제안하였다. Negative Tuple이란 윈도우를 갱신하면서 제거되어야 할 튜플을 나타내는 일종의 더미 튜플(dummy tuple)이다.

그림 5는 Negative Tuple의 작동 방식을 나타낸 것이다. 만일 튜플 6과 튜플 3이 윈도우를 벗어나게 되어 제거되어야 한다면, 6과 3은 Negative로 되어 다시 조인 연산자의 입력으로 들어간다. 튜플 3과 매치되는 것이 없기 때문에 이에 대한 결과 Negative Tuple은 생

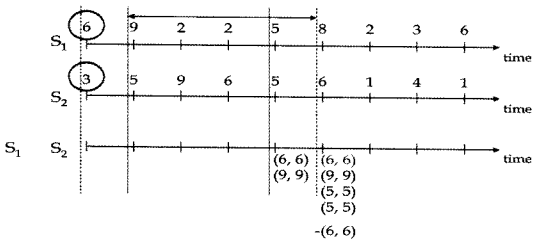


그림 5 Negative Tuple 기법

성되지 않으며 튜플 6에 대한 Negative Tuple은 -(6, 6)이 생성된다. 이 윈도우가 갱신되기 전에 (6, 6), (9, 9)의 결과 튜플이 나와 있기 때문에, Negative Tuple인 -(6, 6)과 동일한 키와 속성을 가지는 (6, 6) 튜플이 제거된다. 다시 말해서, 값 기반 탐색(value-based search)을 통해 Negative Tuple과 동일한 튜플을 찾아 제거한다. 이 기법은 타임스탬프 기반의 윈도우 갱신 기법이 모든 결과 튜플에 대한 검색을 거쳐야 한다는 비효율성을 개선시키는 것과 동시에, 튜플 개수 기반 윈도우가 사용되는 환경에 적용 가능하므로 효율성과 확장성의 측면에서 바람직한 윈도우 갱신 기법이라고 평가할 수 있다. 하지만 Negative Tuple 기법은 정확성 측면에서 문제점을 보인다.

그림 6은 입력 R과 입력 S를 조인할 때의 상황을 나타낸 것으로 괄호 안의 숫자는 튜플이 입력된 차례를 나타낸 것이다. 예를 들어, <120, 30%(1)>은 가장 먼저 입력된 튜플이다. State R에서 가장 상위에 있는 튜플 <120, 25°C(2)>이 윈도우가 갱신됨에 따라 제거되어야 한다면 결과 Negative Tuple은 <120, 25°C, 120, 30%>, <120, 25°C, 120, 35%>이 된다. 올바른 갱신을 수행하는 경우에, State RS에서 <120, 25°C, 120, 30%(3)>과 <120, 25°C, 120, 30%(11)>이 제거되어야 하지만 Negative Tuple 기법은 값 기반 탐색을 수행하므로 <120, 25°C, 120, 30%(7)>과 <120, 25°C, 120, 30%(13)>도 제거된다. 이 결과는 올바르지 못한 결과이다.

이와 같은 값 기반 탐색은 정확하지 못한 결과를 초래하므로, 정확한 튜플을 찾아낼 수 있는 탐색방법이 필요하다. 본 연구에서는 이를 위해 인덱스 할당기법을 제안한다. 인덱스 할당 기법의 자세한 소개는 4장에서 다루도록 한다.

3.3 공유 조인 연산자를 위한 라우팅 기법에 관한 연구

[17]에서는 새로운 튜플이 A와 B에 각각 입력되었을 때, 새로 입력된 튜플 각각 가장 큰 윈도우 크기에 대해 한번씩 매치를 수행하는 LWF(Largest Window First) 기법, 새로 입력된 튜플들에 대해 아직 매치가 완료되지 않은 가장 작은 윈도우 영역을 번갈아 가면서 수행하는 SWF(Smallest Window First), 새로 입력된 튜플들에 대해 가장 많은 질의에 가장 많은 튜플을 먼저 내보낼 수 있는 MQT(Maximum Query Throughput)기법을 제안하였다.

[18]에서는 조인 연산자들을 공유할 때, 윈도우를 기준으로 미리 조인 연산자를 나누고 이를 점진적으로 합쳐나가는 전략(State-Slice)을 사용한다. 이러한 LWF, SWF, MQT, State-Slice Sharing 기법 모두는 원시 스트림만을 입력으로 가지는 환경에서만 작동된다. 특히 State-Slice Sharing 전략은 각 질의가 입력 스트림에 대해 같은 윈도우를 사용하지 않는 환경에서는 적용할 수 없다.

4. MMJoin의 공유 처리

4.1 최적의 전역 공유 질의 실행 계획 수립 근사화 기법

2장에서 언급하였듯이, MMJoin 연산자에 대한 최적의 전역 공유 질의 실행 계획을 수립하는 문제는 NP-Hard이다. 또한 본 연구는 데이타 스트림 환경을 가정하므로 이 문제를 다룰 때 빠른 계산 속도로 최적의 해에 근접하는 근사화 접근 방법이 필요하며 다음과 같은 조건을 만족시켜야 한다.

- 탐색 공간을 줄이고, 적은 탐색으로도 만족할만한 성능 향상을 이끌어내야 한다.
- MJoin연산자의 특성이 충분히 반영되어야 한다.
- 데이타 스트림 환경에서 한정된 메모리만을 사용하게 해주는 슬라이딩 윈도우의 제약사항을 반영해야 한다.

이 조건들을 충족시키기 위해 다음과 같은 관찰들을

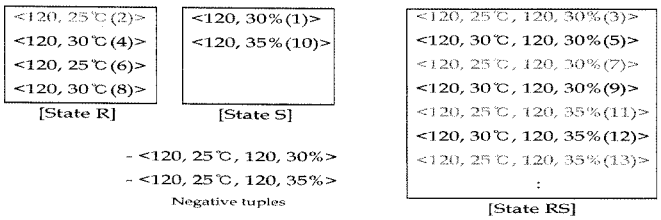


그림 6 Negative Tuple 기법의 비정확성

해 볼 수 있다.

첫째, 모든 가능한 질의 실행 계획들의 조합이 아닌 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. 전역 공유 질의 실행 계획을 수립하여 반환되는 연산자의 수는 본래 독립적으로 수행되는 MJoin이 가진 연산자의 수와 같거나 작은 수가되도록 하는 것이 바람직하다. 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 조인연산자의 비용 모델(cost model)을 다음과 같이 정의한다.

정의 4.1. 비용 모델(cost model)

어떤 MJoin 연산자가 n개의 입력 $R=\{R_1, R_2, ..., R_n\}$을 사용하고 각 R_i의 입력 속도와 윈도우 크기를 각각 r_i, W_i라 하자. 또한 f를 조인 연산에 대한 선택 비율 요소(Selectivity Factor)라 하면, 다음과 같이 비용 모델을 정의할 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 예측할 수 있다. 이를 통해 상위 조인 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, 이는 결국 연산자가 공유된 이후에 독립적인 연산자로 처리하는 경우보다 더 많은 튜플을 저장하게 되는 가능성을 지닌다. 그림 7을 바탕으로 자세하게 설명하면 다음과 같다.

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, 입력 R에 대해서는 k, 입력 S에 대해서는 m만큼의 윈도우 크기를 가지게 된다. 따라서 공유된 연산자에서 처리하는 튜플의 최대 양은 k×m이 된다. 따라서 연산자를 공유하기 전에 다음과 같은 조건에 부합하는지를 평가하여야 한다.

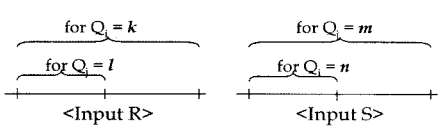


그림 7 두 입력에 대한 서로 다른 윈도우

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. 그렇다면 다음과 같은 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 MMJoin 연산자의 최적의 전역 질의 실행 계획을 근사화하여 수립하는 경험론적 욕심쟁이(heuristic greedy) 알고리즘을 제안한다. 각 질의는 하나의 연산자로 표현되며, 각 연산자는 입력 스트림의 집합으로 표현될 수 있다.

그림 8의 알고리즘에서 보이는 바와 같이 매 단계마다 하나의 질의, 즉, MJoin을 선택한다. 각 단계에서는 가장 많이 공유되는 집합이 복수개가 선택된다면 그 가운데에 가장 많은 처리 비용을 가지는 하나의 집합을 선택한다. 만일 선택된 집합이 정의 4.1의 공유 조건 식에 만족하지 않는다면, 선택 집합을 제외하고 다음 단계로 넘어간다. 만일 선택된 집합이 공유 조건 식에 부합하면, 선택된 집합을 포함하여 모든 주어진 집합에서 선택된 집합의 원소를 포함하는 부분을 찾아 해당 부분을 선택된 집합으로 치환한다. 이렇게 되면 공통 되는 원소들은 하나의 원소로 치환된다. 마지막으로, 하나의 원소만을 가지는 집합을 다음 단계에서 제외한다. 이는 하나의 원소를 가지는 집합은 이미 질의 실행 계획이 완성되었음을 의미하기 때문이다. 이러한 작업을 모든 질의가 제외될 때까지 반복한다.

예를 들어 설명하면 다음과 같다. 설명의 간소화를 위해 모든 조인에 대한 선택 비율이 0.1이고 모든입력 스트림에 대한 윈도우의 크기가 1000 rows로 하여 공유 조건 식에 모두 만족된다고 가정하자.

그림 9와 같이 질의가 주어지면 먼저 가장 많은 질의에 포함되는 것들을 고른다. 위 예에서는 Q_1, Q_2, Q_3이 모두 Q_4, Q_5에 포함된다. 조인 연산자의 처리 비용은 윈

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

그림 8 경험론적 욕심쟁이 전략을 이용한 MMJoin의 최적화 알고리즘

```
Q1 = R[Rows 100], S[Rows 100] = {R[100], S[100]}
Q2 = R[Rows 100], T[Rows 100] = {R[100], T[100]}
Q3 = S[Rows 100], T[Rows 100] = {S[100], T[100]}
Q4 = R[Rows 100], S[Rows 100], T[Rows 100] = {R[100], S[100], T[100]}
Q5 = R[Rows 100], S[Rows 100], T[Rows 100], U[Rows 100] = {R[100], S[100], T[100],
                                                          U[100]}
```

그림 9 예제 질의들

도우의 크기와 선택비율에 비례하기 때문에 Q_1, Q_2, Q_3 모두 같은 처리 비용을 가지게 된다.[1] 본 예에서는 Q_1이 가장 먼저 발생하였다고 가정하고 Q_1을 선택하기로 한다. Q_1이 선택되면 Q_1을 포함하는 집합에서 공통되는 원소들을 집합 Q_1으로 치환한다. 그 결과는 다음 그림 10과 같다.

그림 10에서 Q_1은 하나의 원소만을 가지므로 질의 실행 계획이 완성되었음을 의미한다. 따라서 첫 번째 단계

```
Q1 = {{R[100], S[100]}}
Q2 = {R[100], T[100]}
Q3 = {S[100], T[100]}
Q4 = {{R[100], S[100]}, T[100]}
Q5 = {{R[100], S[100]}, T[100], U[100]}
```

그림 10 알고리즘의 첫 번째 단계를 수행 한 후의 예

1) 대부분, 윈도우의 크기와 선택 비율이 다르기 때문에 같은 처리 비용을 가지는 경우는 매우 드물다.

```
Q1 = {{R[100], S[100]}}
Q2 = {R[100], T[100]}
Q3 = {S[100], T[100]}
Q4 = {{R[100], S[100]}, T[100]}}
Q5 = {{{R[100], S[100]}, T[100]}, U[100]}
```

그림 11 알고리즘의 두 번째 단계를 수행 한 후의 예

이후에 Q_1은 제외된다. 다시 가장 많은 집합에 포함될 수 있는 집합을 선택하면 Q_4만이 Q_5에 포함되므로 Q_4를 선택한다. 이 때 하나의 집합만이 선택되었으므로 처리 비용을 고려할 필요가 없다. Q_4를 포함하는 집합에 공통되는 원소들을 Q_4의 집합으로 치환하면 다음 그림 11과 같다.

이제 Q_4는 하나의 원소만을 가지게 되므로 다음단계에서 제외되며, 나머지 집합 Q_2, Q_3, Q_5는 서로 간의 포함관계를 가지지 않기 때문에 각각의 질의 실행 계획을 수립하게 된다. 최종적인 전역 질의 실행 계획은 그림 12와 같다.

Q_1 = {{R[100], S[100]}}
Q_2 = {{R[100], T[100]}}
Q_3 = {{S[100], T[100]}}
Q_4 = {{R[100], S[100]}, T[100]}}
Q_5 = {{{{R[100], S[100]}, T[100]}, U[100]}}

그림 12 알고리즘의 최종 단계를 수행 한 후의 예

MMJoin 기법은 경험론적 욕심쟁이 알고리즘을 사용하는데, 각 단계마다 원소를 대상으로 공유가 되는 것이 아닌, 질의 집합(MJoin 연산자)을 기준으로 공유를 진행한다.

정리 4.1. MMJoin의 최적화 알고리즘이 반환하는 연산자 수를 $NumO*$, 독립적으로 수행되는 본래의 연산자 수를 $NumO$라 하면 $NumO \geq NumO^*$는 항상 만족한다.

증명. MMJoin 기법은 질의 집합을 기준으로 평가한다. 따라서 매 단계에서 적어도 하나의 질의 집합이 제외된다. MMJoin 기법이 모든 질의가 제외되는 시점에 완료되기 때문에, 총 반복 횟수는 MMJoin 기법이 반환하는 연산자의 총 수와 같다. 위 예에서는 총 6번의 단계를 거치게 되어 본래의 연산자 수와 같은 6개의 연산자를 반환하였다. 최악의 경우에는 모든 질의가 서로 포함관계를 가지지 않으므로 매 단계마다 하나의 질의 집합만이 제외된다. 이는 결국 본래의 연산자 수인 $NumO$와 동일한 연산자 수를 반환함을 말한다. 만일, 매 단계에서 공유되는 질의 집합이 하나 이상 발생하면, 적어도 하나 이상의 질의가 다음 단계에서 제외되며, 이는 결국 MMJoin 기법의 반복 횟수가 본래의 질의 수 보다 같거나 작아지는 것을 의미한다. 따라서 MMJoin 기법이 반환하는 연산자의 총 수는 본래의 연산자 수와 같거나 작다.

연산자의 수를 줄이게 되면, 스케줄러에 의한 문맥 교환을 줄일 수 있기 때문에 어느 정도의 성능 향상을 기대할 수 있다. 다음은 MMJoin 기법의 시간 복잡도를 분석하여 데이타 스트림 환경에 적용 가능함을 보인다.

그림 8의 알고리즘에서 입력으로 질의 집합이 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 각 입력마다 최대 크기의 윈도우를 찾아야 하며 모든 질의의 윈도우를 조사해야 하므로 $O(nm + nm) = O(nm)$의 시간이 소요된다. 따라서, 매 단계에서는 $O(nm^2 + nm)$의 시간 복잡도를 가진다. 최악의 경우 제안한 알고리즘은 매 단계에서 하나의 질의만을 제외시키므로 n번 반복된다. 따라서 MMJoin 기법의 총 시간 복잡도는 $O(n^2m^2 + n^2m + n^2)$이 된다. 이는 최적화 시간을 다항시간(polynomial time)으로 줄이게 되었음을 말한다. 결론적으로, 전통적인 다중 질의 최적화 기법이 최악의 경우 모든 탐색 영역(모든 가능한 질의 실행 계획들의 조합)을 조사해야 하므로, MMJoin 기법이 더욱 빠른 속도로 전역 질의 실행 계획을 수립함을 알 수 있다.

4.2 조인 연산 결과를 위한 윈도우 갱신 및 라우팅 기법

4.2.1 인덱스 할당

원시 스트림에서 윈도우 갱신에 의해 튜플이 제거되거나 라우팅 정보가 추가될 때마다 이 튜플과 관련된 모든 조인 연산 결과 튜플들에 반영되어야 정확한 윈도우 갱신과 라우팅을 수행할 수 있다. 이를 위한 방안으로 인덱스 할당 기법을 제안한다. 즉, 인덱스를 할당함으로써 각 튜플이 어떠한 튜플들에 의해 생성된 것인지, 혹은 어떠한 튜플들을 생성하였는지를 파악할 수 있도록 해준다. 이를 위해 각 튜플에 PrevID와 PostID의 두 가지 인덱스 정보를 기입해 준다.

그림 13은 제안하는 인덱스 스킴을 나타낸 것으로 각 튜플이 PrevID와 PostID의 쌍을 추가적으로 가짐을 보인다. PrevID는 이 튜플이 어떠한 튜플들로부터 생성된

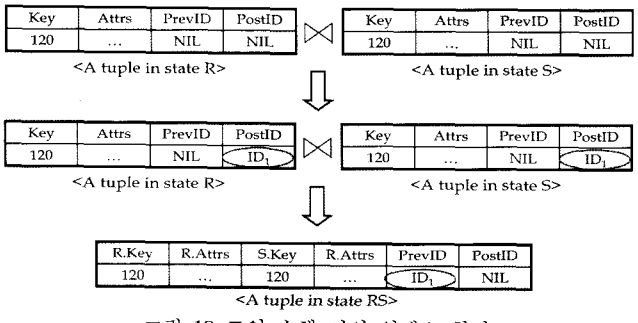


그림 13 조인 수행 시의 인덱스 할당

것인지를 알려주며, PostID는 이 튜플이 어떠한 튜플들을 생성한 것인지를 알려주게 된다. 튜플이 처음 시스템에 입력되면, 그 튜플은 다른 튜플로부터 조인되어 생성된 것이 아니므로 PrevID는 NULL값을 가지며, 아직 조인 연산이 수행되지 않았으므로 PostID의 값도 NULL을 가진다. 입력 R과 S는 원시 입력 스트림을 의미하므로 그림 13에서와 같은 NULL 인덱스 값을 가진다. R과 S의 조인이 수행된다면, 해당하는 두 튜플의 PostID에 동일한 인덱스 값을 추가한다. 그림 13에서는 이를 고유한 인덱스 값으로 ID1을 추가하였다. 두 튜플에 의해 생성된 결과 튜플 역시 PrevID와 PostID 속성을 가지게 되는데, 조인에 참여한 튜플이 매치되면서 추가된 인덱스 값과 동일한 값이 조인 연산 결과 튜플의 PrevID에 기입된다. 이렇게 함으로써 조인 연산 결과 튜플이 어떤 튜플들에 의해 생성된 것인지를 알려주며, 각 튜플이 어떤 튜플들을 생성하였는지를 알려주게 된다. 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.

4.2.2 라우팅을 위한 Dead Vector

각 튜플에 보다 세밀한 위치 정보를 나타내기 위해서는 인덱스를 할당하는 작업과 더불어 라우팅에 대한 정보를 추가하여야 한다. 라우팅은 결국, 조인 연산자가 공유되었을 때, 각 질의에서 각각의 입력에 대해정의한 윈도우를 기반으로 조인 연산 결과 튜플들이 어떠한 질의에 만족되고, 어떠한 질의에 만족되지 않는가를 평가하는 작업이라고 말할 수 있다. 따라서 조인 연산 결과 튜플이 연산자의 외부로 내보내지기 전에 분기 정보를 계산하여야 한다. 본 연구에서는 [19]에서는 제안한 DeadVector 방법을 변형하여 적용한다. Dead Vector 기법은 본래 조인 연산과 선택(selection) 연산의 공유처리를 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. 본 연구에서는 Dead Vector를 이용하여 라우팅을 위한 정보를 사전에 튜플 내에 기입하고, 조인 연산 후에 이를 이용하여 각각의 질의로 분기시키는 방법을 제안한다. 이를 자세하게 설명하면 다음과 같다.

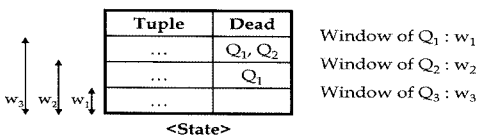
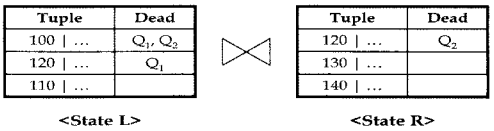


그림 14 Dead Vector의 예

그림 14는 서로 다른 윈도우 크기를 사용하는 3개의 질의가 하나의 연산자로 공유되어 처리 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 상황 저장소(state)에 저장한다. 처음에 입력된 튜플은 w_1, w_2, w_3의 윈도우 내에 모두 포함되어 있으므로, 이 튜플로 인해 생성된 조인 연산 결과 튜플은 모든 질의로 내보내진다. 만약 다른 튜플이 입력되면 기존의 튜플은 w_1(1개의 튜플)을 벗어나게 되어 Dead Vector에는 Q_1을 기입하는데, 이것은 해당 튜플이 Q_1에 만족하지 않는다는 것을 나타내는 정보이다. 또 다른 튜플이 입력되는 두 튜플은 계속 위로 올라가서 가장 처음에 입력된 튜플은 w_2(2개의 튜플)를 벗어나며, 그 다음으로 입력된 튜플은 w_1(1개의 튜플)을 벗어나게 된다. 따라서 가장 처음 입력된 튜플의 Dead Vector에는 Q_2가 추가되고, 그 다음의 튜플에는 Q_1이 Dead Vector에 추가된다. 이 정보는 추후 조인 연산 결과에 대한 라우팅 정보로 활용된다.

그림 15는 조인 연산 결과 튜플이 어떻게 Dead Vector를 가지는가를 나타낸 것이다. 만일 입력 스트림 L과 R이 조인된다면, 상황 L과 상황 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 되는데, 그 결과는 그림 15(b)의 튜플과 같다. 즉, 상황 L의 튜플이 Dead Vector로 가지고 있던 Q_1과 상황 R의 튜플이 Dead Vector로 가지고 있던 Q_2를 병합하여 조인 연산결과 튜플의 Dead Vector에 Q_1, Q_2를 기입한다. 이 정보는 조인 연산 결과 튜플이 Q_1, Q_2에 모두 만족되지 않으므로 Q_1, Q_2를 제외한 다른 질의들로 내보내져야 함을 의미한다.



(a) 입력 스트림 L과 R의 Dead Vector

Tuple L	Tuple R	Dead
...	...	Q_1, Q_2

(b) 조인 연산 결과 튜플의 Dead Vector

그림 15 Dead Vector의 사용 예

4.2.3 Purging Tuple과 Dead Tuple

앞선 절에서는 원시 스트림에 대해 인덱스와 Dead Vector를 할당하는 기법을 소개하였다. 원시 스트림의

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 16 Purging Tuple 생성의 예

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 17 Dead Tuple 생성의 예

튜플에서 어떤 튜플이 제거되거나 튜플에 새로운 Dead Vector 값이 추가될 때마다 이와 관련된 상위 조인 연산자의 연산 결과튜플에도 영향을 끼쳐야 한다. 본 연구에서는 그러한 작업을 하는 것으로 Purging Tuple과 Dead Tuple의 생성과 사용을 제안한다. 이 두 튜플은 3장에서 언급한 Negative Tuple 기법을 변형한 형태로, Negative Tuple이 값 기반 탐색을 하는 점과 Purging Tuple및 Dead Tuple이 인덱스 기반 탐색을 수행한다는 점에서 차이점을 보인다. 또한 Purging Tuple 및 Dead Tuple을 생성하기 위해 다시조인 연산을 거치지 않는다는 점도 다르다.

• Purging Tuple의 생성

인덱스 값을 통해 각 튜플은 어떤 튜플들에 의해 생성되며, 어떠한 튜플을 생성했는지에 대한 정보를 유지한다. 이러한 정보를 이용하여 윈도우 갱신 시 튜플이 제거되었을 때 이와 연관된 튜플을 삭제해 주어야 한다. 이를 위해 생성되는 튜플이 Purging Tuple이다. Purging Tuple은 인덱스 정보와 현재 튜플이 가상의 튜플임을 알려주는 특별한 플래그(Flag)[2]를 가지며 실제 조인 연산에는 참여하지 않고 튜플을 삭제하는 데에만 사용된다.

그림 16은 윈도우갱신에 의해 제거되는 튜플과 그에 따라 생성되는 Purging Tuple의 예를 보여준다. 그림 16(a)에서 PostID는 세 개의 인덱스 값을 가지고 있는데, 이는 이 튜플이 지금까지 세 개의 조인 연산 튜플을 생성하였다는 것을 의미한다. 또한, 그림 16(a)의 튜플은 Purging Tuple에 의해 윈도우가 갱신되는 것이 아닌, 원시 스트림에 대한 윈도우 갱신 시 제거되는 튜플임을 주의하자. 따라서 이 PostID에 포함된 인덱스들 중 동일한 PrevID를 가지는 상위 조인 연산자의 튜플들을 제거해 주어야 하는데, Purging Tuple은 해당 PostID 값을 상위로 전달해주는 역할을 한다. 그림 16(b)는 그림 16(a)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.

• Dead Tuple의 생성

어떠한 원시 스트림튜플의 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 PostID 값을 얻고, 인덱스 해시 테이블을 이용하여 조인 상황 테이블의 같은 PrevID를 가진 튜플들을 탐색하고, 매치되는 튜플에 Dead Tuple이 가진 Dead Vector를 결합시킨다. 그림 17는 이러한 Dead Tuple의 예를 보여준다. 그림 17(a)에서와 같이 어떤 원시 스트림 튜플이 Dead Vector값으로 Q1이 추가되면, 해당 튜플로 인해 생성된모든 조인 연산 결과 튜플에도 전달이 되어야 한다. 그림 17(b)가 이 때 생성되는 Dead Tuple을 나타낸 것이다. Dead Tuple은 Purging Tuple과 다른 플래그를 가지는데, 본 연구에서는 이를 이중 Asterrisk(**)로 표시한다.

4.2.4 인덱스 해시 테이블

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. 이러한 탐색을 별도의 자료구조 없이 진행하게 되면 조인 연산자 내의 모든 튜플은 인덱스를 기준으로 평가된다. 본 연구에서는 인덱스 기반의 탐색을 보다 빠르게 수행하도록 인덱스 해시 테이블(Index Hash Table)을 조인 연산자 내에 구성한다. 인덱스 해시 테이블은 해당 조인 연산자 내의 튜플들이 가지고 있는 PrevID를 기반으로 구성된다. 따라서 Purging Tuple 및 Dead Tuple이 들어왔을 때 원하는 PrevID를 빠르게 찾을 수 있도록 해준다. 그림 18은 인덱스 해시 테이블의 구조 및 본 연구에서 가정하는 조

2) 특정 속성에 Asterisk(*) 문자 등을 사용

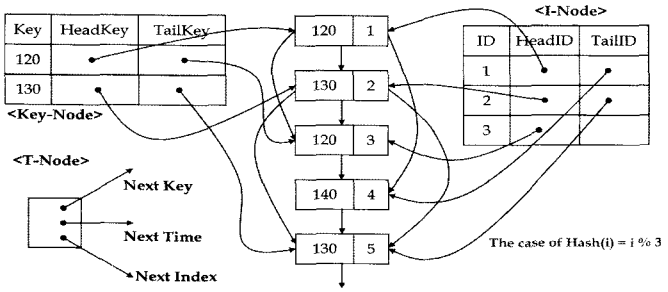


그림 18 조인 연산자의 자료구조

인 연산자의 자료구조를 나타낸 것이다.

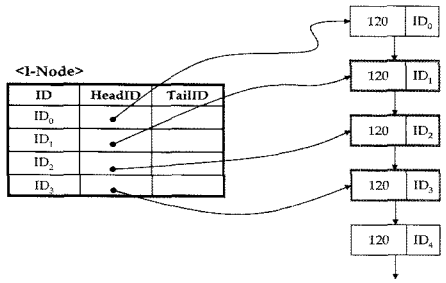
그림 18에서 <I-Node>는 인덱스 해시 테이블의 헤더(Header)정보로 각 PrevID의 값이 가장 처음으로 발생되는 튜플과 마지막에 나타나는 튜플에 대한 포인터(Pointer)를 유지한다. 각 튜플이 저장되는 자료구조는 <T-Node>로 표현되는데, 이들은 같은 PrevID를 가지는 다음 튜플에 대한 포인터를 유지함으로써 인덱스 해시 테이블의 계속적인 탐색을 가능하게 한다.

4.2.5 Purging/Dead Tuple의 탐색

Purging Tuple 및 Dead Tuple이 생성되어 상위 조인 연산자로 보내지면 상위 조인 연산자에서는 Purging Tuple 및 Dead Tuple의 PostID 값을 읽고 그 값들 중 하나에 해당되는 튜플을 찾아 제거하거나 새로운 Dead Vector 값을 추가하여야 한다. 그림 19(a)는 윈도우 갱신에 의해 제거되는 튜플이나 새로운 Dead 값이 추가된 튜플로 인해 생성된 Purging Tuple 및 Dead Tuple을 나타낸 것이다. 이 Purging/Dead Tuple은 상위 조인 연산자에 도착하면 PostID 값을 추출하고 그림 19(b)와 같이 인덱스 해시 테이블을 통해 이와 해당 PrevID값을 가지는 튜플들을 찾게 된다.

Key	Attrs	PrevID	PostID	Dead
120	*(**)	NIL	ID_1, ID_2, ID_3	Q_3

(a) Purging tuple 및 Dead Tuple



(b) 인덱스 해시 테이블 탐색

그림 19 Purging Tuple 및 Dead Tuple에 의한 인덱스 해시 테이블 탐색

그림 20의 알고리즘은 지금까지 설명한 인덱스 할당, Dead Vector의 사용, Purging Tuple 및 Dead Tuple 기법을 이용한 튜플 유효화 알고리즘을 나타낸 것이다.

5. 성능 평가

본 연구에서는 조인 연산 시 여러 입력을 취하는 MJoin연산자 환경에서 여러 조인 질의들이 각각 하나의 연산자로 독립적으로 처리되었을 때의 비효율적인 측면을 설명하고 있다. 그러나, MJoin을 독립적으로 각각 수행했을 때에는 연산자가 공유되어 있지 않기 때문에, 모든 윈도우 갱신이 원시 스트림에 대해서만 수행되며 라우팅이 필요하지 않다. 제안하는 MMJoin 기법은 연산자를 공유시킴으로써 조인 트리가 형성되고, 그 가운데에 조인 연산 결과에 대한 윈도우 갱신과 라우팅을 위해 Purging Tuple과 Dead Tuple을 사용한다.

본 장에서는 MJoin 연산자를 공유하는 MMJoin 기법이 추가적인 비용에도 불구하고 독립적으로 수행하는 MJoin 기법보다 성능이 우수함을 보인다.

5.1 실험 환경 설정

본 실험에서 MJoin과 제안하는 MMJoin 기법은 모두 자바로 구현하였으며, 펜티엄 IV 3.0Ghz 프로세서와 1GB DDR2 메모리가 장착된 윈도우 운영체제에서 자바 가상 머신 1.5.0을 이용하였으며 별도의 쓰레드(Thread)를 생성하여 성능 결과를 측정하였다.

• 실험 변수 및 고정 값 생성

데이타 스트림에서 각 스트림에 대한 튜플들은 동일한 스키마를 가진다. 전통적인 관계 형 데이타베이스 시스템에서 각 스트림의 전체 튜플 집합은 릴레이션과 동일한 의미로 볼 수 있다. 본 실험에서 사용되는 모든 스트림 튜플의 스키마는 표 1과 같다.

실험에서는 20개의 스트림을 가정하고, 각 스트림의 입력 속도를 초당 300개 튜플(300tuples/sec)로 정의하였다. 모든 튜플에서 키 속성 값은 0부터 1000까지의 정수 값을 가지도록 하였다. 이 키 속성 값은 모든 튜플에 걸쳐 같은 빈도수로 발생하도록 하였는데, 발생 순서

```
Input : a input tuple, tuplenew
if tuplenew is Purging Tuple then
begin
    for each id ∈ PostID values in tuplenew do
    begin
        remove a tuple with PrevID value = id from correlated State
        generate Purging Tuple tuplepurging for a removed tuple
    end
end
else if tuplenew is Dead Tuple then
begin
    for each d ∈ Dead values in tuplenew do
    begin
        for each id ∈ PostID values in tuplenew do
        begin
            add d in a tuple with PrevID value = id as Dead value
            generate Dead Tuple tupledead for the tuple
        end
    end
end
else
begin
    for each w ∈ Window sizes in join operator do
    begin
        if oldest tuple in state is out of w
            add Dead in the tuple for w and generate Dead Tuple tupledead
    end
    if oldest tuple in state is out of maximum window size
        generate Purging Tuple tuplepurging
end
```

그림 20 조인 연산을 위한 유효화 알고리즘

표 1 튜플 스키마

Key	Data	PrevID	PostID	Dead	Timestamp

는 무작위(Random)로 하여 슬라이딩 윈도우 내에 있는 키 속성 값들이 매번 바뀌므로 조인 연산자의 선택비율도 계속적으로 변화한다.

이렇게 생성된 입력 스트림에 대해 여러 질의들이 조인을 수행하는데, 각 질의는 최대 20개의 서로 다른 입력 스트림을 사용할 수 있다. 실험의 정교함을 위해 여러 질의에 걸쳐 나타나는 입력 스트림들의 비대칭도(skewness)를 반영하였다. 본 실험에서는 전체 질의에서의 입력 스트림에 대한 비대칭도를 지프 분포(Zipf distribution)[20]를 이용하여 적용하였다. 또한, 각 질의에서 각각의 입력 스트림에 대해 500개의 튜플(500 rows), 1000개의 튜플(1000 rows), 1500개의 튜플(1500 rows)중 하나를 무작위로 선택하여 윈도우로 사용하였으며, 각 윈도우는 균등 분포로 발생하도록 하였다. 세 가지 윈도우는 MMJoin 기법에서 사용하는 공유 조건식에 부합하지 않는 경우가 발생할 수 있도록 하기 위함이다. 표 2는 본 실험에서 사용하는 실험 매개 변수를 나타낸 것이다.

5.2 실험1: 전역 공유 질의 실행 계획 수립 시간 비교

실험 1에서는 실험 매개변수 SO를 0.5로 고정한다. 그림 16은 각 질의에서 나타나는 입력 스트림의 비대칭도가 0.5일 때, 독립적으로 수행되는 MJoin과 MMJoin 기법을 수행하기 위한 최적화 시간 및 연산자 할당 시간을 보여준다. 전역 공유 질의 실행 계획 수립 시간의 측정은 다음과 같다.

실험측정 단위. 사용자가 입력한 여러 질의들에 대한 연산자 할당 시간을 수행하는 시간 t_{assigh}, 그림 8의 알고리즘에 의한 계산 시간을 $t_{optimizing}$이라 하자. 그럼 전역 공유 질의 실행 계획 수립 시간 MQST(Multiple-Query Setup Time)는 $t_{assign} + t_{optimizing}$ 이다.

그림 8의 알고리즘은 연산자의 수가 줄어드는 효과를 가져오기 때문에 MJoin을 독립적으로 수행하기 위한 본래의 연산자 할당 시간보다 약간 적은 시간이 소요된다. 하지만, MMJoin 기법에서는 질의 수가 증가할수록 별도의 최적화를 위한 연산이 증가하여 더 많은 시간이 요구된다.

표 2 입력 스트림과 질의에 대한 실험 변수

Parameter	Range	Description
Q	10 to 100	The total number of distinct queries
SO	0 to 1	Skewness of input streams over queries
WS	500 to 1500	Window sizes for input stream

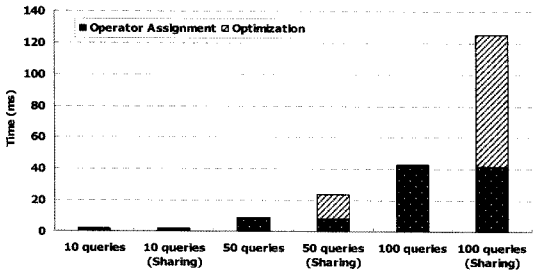


그림 21 질의 수 변화에 따른 수립 시간 비교 (SO=0.5)

그림 21에서의 측정 시간 단위는 ms(milli-second)인데, 이 시간은 빠른 처리가 요구되는 데이타 스트림 환경에서 비교적 큰 단위라고 볼 수도 있다. 그러나 본 실험에서 성능 측정을 위해 구현된 질의 최적화와 연산자 할당은 문자열(string)을 기반으로 수행하였기 때문에 실제 데이타 스트림 관리 시스템내의 구문 분석(parsing) 엔진으로 내장된다면 더 빠른 시간 내에 최적화를 수행할 수 있다.

5.3 실험2: MJoin과 MMJoin 기법의Throughput

그림 22(a)는 SO를 0.5로 고정시키고 질의의 수를 변화시켰을 때의 평균 처리량을 보여준다. 평균 처리량에 대한 측정 단위는 다음과 같다.

실험측정 단위. 총 모든 질의가 특정 시점 i초에서 $i+1$초 사이에 내보내는 결과 튜플 수의 총 합을 $throughput_i$, 측정한 총 시간을 t_{total}, 질의의 총 수를 Q라 하면, 각 질의에 대한 평균적인 처리량은 AT는 $\sum_{i=0}^{t_{total}} throughput_i/(t_{total} \cdot Q)$ 이다.

질의의 수가 많아질수록 독립적으로 수행되는 MJoin의 처리량보다 MMJoin 기법의 처리량이 증가하는 것을 볼 수 있는데, 이는 질의의 수가 증가할수록 연산자가 더 많이 공유될 수 있는 확률도 증가하기 때문이다. 그림 22(b)는 질의의 수를 100개로 고정하고 입력 비대칭도를 변화시켰을 때의 평균 처리량을 보여준다. 입력 비대칭도가 증가할수록 여러 질의들 가운데에 공통되는 부분도 증가하여 더 많은 공유가 일어나 MMJoin 기법의 상대적인 평균 처리량도 증가함을 볼 수 있다. 결국, 질의의 수와 입력 스트림의 비대칭도가 증가한다면 각 질의들이 포함관계를 가지는 경우가 증가하므로 MMJoin 기법에서 공유되는 연산자의 수도 증가하게 된다.

그림 22의 두 그래프에서 일정 수준 이상 질의 수나 입력 비대칭도에 다다르면 큰 처리 향상률의 증가를 보이지 않는데, 이는 일정 수준의 질의 수와 입력 비대칭도가 주어지면 이미 연산자가 충분히 공유될 수 있는 환경을 갖추기 때문이다.

5.4 실험3: MJoin과 MMJoin 기법의 튜플 사용량

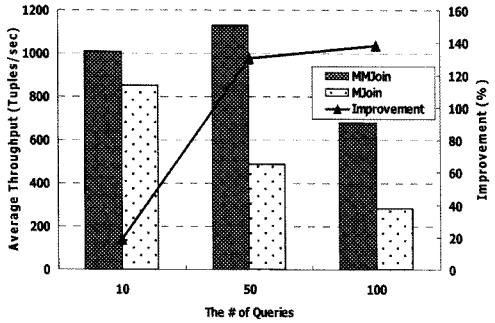
그림 23은 SO를 0.5로 고정시키고 질의의 수를 변화시켰을 때와 질의의 수를 100개로 고정하고 SO를 변화시켰을 때의 평균적인 튜플 사용량을 보여준다. 튜플 사용량에 대한 측정 단위는 다음과 같다.

실험측정 단위. 하나의 조인 연산자 op_i의 스테이트에 있는 모든 튜플의 수를 st_i, 전체 조인 연산자의 총 수를 op_{total}이라 하자. 그럼 어떤 시점에서 시스템에 등록된 모든 조인 연산자가 메모리 내에서 유지하고 있는 모든 튜플의 수는 $\sum_{i=1}^{op_{total}} st_i$ 이다.

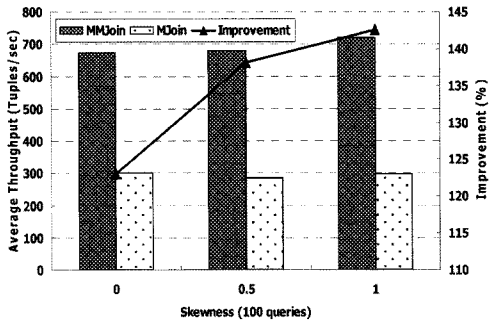
질의의 수가 증가할수록, 입력 비대칭도가 증가할수록 MMJoin 기법이 사용하는 상대적인 튜플 사용량이 점차 감소하는 것을 알 수 있다. 이는 앞선 실험에서 설명하였듯이, 질의의 수와 입력 비대칭도가 증가하면 연산자가 더 많이 공유되기 때문이다. 연산자가 공유될 때 사용되는 튜플량의 감소는 본 연구에서 정의한 공유 조건 식에 의해 보장된다.

5.5 실험4: MMJoin 기법의 추가적인 비용

MJoin을 독립적으로 각각 수행했을 때에는 연산자가 공유되어 있지 않기 때문에, 모든 윈도우 갱신이 원시스트림에 대해서만 수행되며 라우팅이 필요하지 않다. 따라서 Purging Tuple과 Dead Tuple에 대한 추가적인

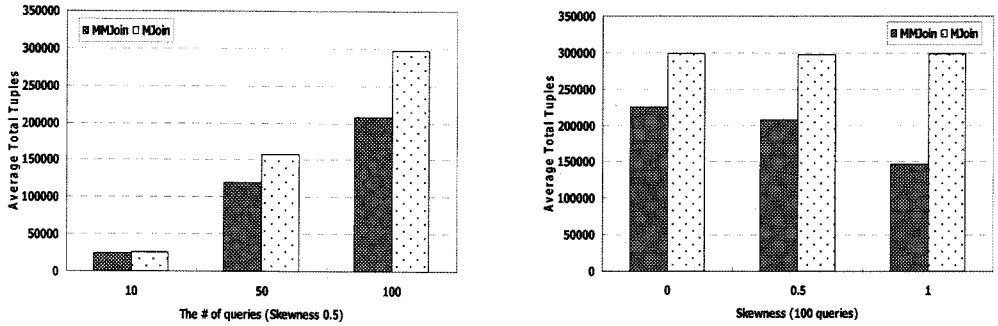


(a) 질의 수에 따른 처리량 비교 (SO=0.5)



(b) Skewness에 따른 처리량 비교 (Q=100)

그림 22 평균 처리량 비교



(a) 질의 수에 따른 튜플량 비교 (SO=0.5) (b) Skewness에 따른 튜플량 비교 (Q=100)

그림 23 평균 튜플 사용량 비교

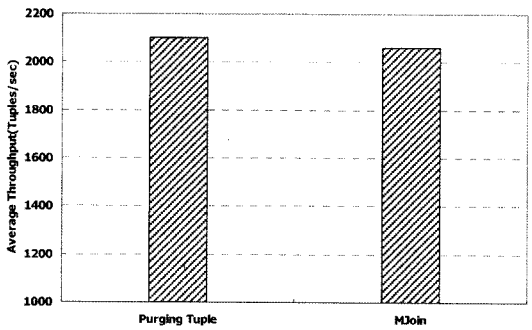


그림 24 평균 처리량: A⋈B⋈C vs (A⋈B)⋈C

계산이 필요하지 않다. 그러나 MMJoin 기법은 연산자를 공유시킴으로써 조인 트리가 형성되고, 그 가운데에 조인 연산 결과에 대한 윈도우갱신과 라우팅을 위해 Purging Tuple과 Dead Tuple을 사용하였다. 이 두 기법은 연산자가 공유되어 얻을 수 있는 이득을 상쇄시킬 수 있을지도 모른다.

그림 24는 모든 입력에 대해 1000개의 튜플을 윈도우로 사용(1000 rows)하는 환경에서 A⋈B⋈C (MJoin)과 (A⋈B)⋈C의 조인 트리가 형성되어 있을 때 Purging Tuple을 사용한 결과를 보여준다. Purging Tuple이 적용된 경우, MJoin 보다 약간 높은 처리량을 나타내는데, 이는 조인 트리가 형성되었을 때, MJoin에서 매번 수행해야 될 A⋈B 계산을 미리캐시(cache)한 효과를 가지기 때문이다. 즉, 연산자를 공유하여 처리하는 데에 따른 계산 이득을 Purging Tuple의 추가적인 계산 비용이 상쇄시킬 만큼 크지 않다는 것이다. Dead Tuple 역시 연산자가 공유되어 있는 상황에서 사용되는 것으로 여러 연산자를 한번의 계산으로 처리하는 이득을 Dead Tuple만의 추가적인 계산으로 상쇄시킬 수 없음은 자명하다. 이는 Dead Tuple이 공유된 연산자에서 기존의 연산자들이 공유된 수에 비례하여 발생하기 때문이다. 즉, Purging Tuple과 Dead Tuple이 MMJoin 기법에서 사용되지만 독립적으로 수행되는 MJoin보다 뛰어난 처리량과 적은 메모리 사용량을 보이는 것으로도 Purging Tuple과 Dead Tuple의 추가적인 계산 비용이 매우 적음을 확인할 수 있다.

6. 결론 및 추후 연구

본 논문은 데이타 스트림 연구 분야에서 다중 조인 질의 영역에서의 효율적인 처리 기법인 MMJoin 기법을 제안하였다. MMJoin 기법들의 특징은 다음과 같다.

첫째, 데이타 스트림에서 전역 공유 질의 실행 계획의 수립 기법 시간을 대폭 줄였다. MMJoin 기법은 질의만을 기준으로 탐색을 수행하며, 슬라이딩 윈도우 제약사항을 충분히 고려하여 데이타 스트림 환경에 적합한 질의 실행 계획 수립을 가능케 하였다. MMJoin 기법이 적은 탐색 대비 높은 성능 향상을 이끌어냄을 실험을 통해 확인하였다.

둘째, 공유 질의 실행 계획 하에서 정확한 윈도우 갱신 기법을 제안하였다. 전역 공유 질의 실행 계획을 수립하면 MJoin이 쪼개져 트리 혹은 그래프의 형태로 변형되어 연산 결과 튜플에 대한 윈도우 갱신 시 문제점이 있었다. 본 논문에서는 시간 기반 윈도우와 튜플 개수 기반 윈도우에 모두 적용될 수 있도록 인덱스 기반 탐색을 제안하였다. 또한 이에 따른 인덱스 할당 기법과 조인 연산자 내부의 상황 자료구조를 확장하였다. 이러한 프레임이 주어졌을 때, 윈도우 갱신에 있어 정확성을 유지하며 적은 탐색만을 지원하는 Purging Tuple 기법을 제안하였다.

셋째, 공유 질의 실행 계획 하에서 정확한 라우팅 기법을 제안하였다. 라우팅은 조인 연산 결과 튜플들에 대한 윈도우 갱신과 밀접한 연관을 가지고 있으므로 윈도우 갱신에서 사용된 인덱스 정보를 활용하여 탐색을 수행한다. 이와 더불어 라우팅에 필요한 추가적인 Dead 속성을 가짐으로써 연산 결과에 대한 빠른 라우팅을 지

원하였다. 조인 연산 결과에 라우팅 정보를 추가하는 Dead Tuple 기법은 Purging Tuple 기법과 매우 유사한 작업을 수행한다. 인덱스를 기반으로 탐색을 수행하기 때문에 해당되는 튜플에 정확하게 라우팅 정보를 추가하게 된다.

그 동안의 데이타 스트림 관리 시스템은 인터넷 로그 분석, 이동 물체 정보 분석 등의 특별한 목적을 가진 시스템으로 간주될 수 있다. 하지만 유비쿼터스와 같이 여러 사용자가 중앙 시스템에 접근하여 정보를 이용할 수 있는 환경에서는 시스템에서 수많은 사용자 질의를 처리해야 된다. 많은 사용자가 접근하는 시스템에서는 데이타 스트림에 대한 보다 빠른 처리가 요구되는데, MMJoin 기법이 이러한 환경에 보다 빠른 처리를 가능케 해줄 것으로 예상된다.

본 논문에서 제안한 기법은 주로 정적인 환경을 가정하였기 때문에 질의가 추가되거나 삭제될 때 전역 공유 질의 실행 계획을 다시 수립해야 하는 문제점을 가지고 있다. 동적인 환경에서 전역 공유 질의 실행 계획을 매우 적은 비용으로 수립할 수 있는 기법에 대한 연구가 필요하다.

참 고 문 헌

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," In Proc. 21st ACM Sym. on Principles of Database Systems, pp. 1-16, 2002.

[2] J. Naughton, D. DeWitt, and D. Maier. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, Vol.24, No.2, pp. 27-33, 2001.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The International Journal on Very Large Data Bases*, Vol.12, Issue 2, pp. 120-139, 2003.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," In Proc. 1st Biennial Conf. on Innovative Database Research, pp. 269-280, 2003.

[5] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards Sensor Database Systems," In Proc. 2th Int. Conf. on Mobile Data Management, pp. 3-14, 2001.

[6] S. Schmidt, M. Fiedler, and W.Lehner, "Source-aware Join Strategies of Sensor Data Streams," In Proc. 17th Int. Conf. on Scientific and statistical database management, pp. 123-132, 2005.

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

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

[9] S. D. Viglas, J. F. Naughton, and J. Burger, "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources," In Proc. 29th VLDB Conf., pp. 285-296, 2003.

[10] L. Golab and M. T. Ozau, "Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams," In Proc. 29th VLDB Conf., pp. 500-511, 2003.

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

[12] L. Ding and E. A. Rundensteiner, "Evaluating Window Joins over Punctuated Streams," In Proc. 13th ACM Int. Conf. on Information and Knowledge Management, pp. 98-107, 2004.

[13] K. Shim and T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, Vol.13, Issue 1, pp. 23-52, 1988.

[14] J. Chen, and D. J. DeWitt, "Dynamic Re-grouping of Continuous Queries," In Proc. 28th VLDB Conf., pp.430-441, 2002.

[15] Y. Watanabe, and H. Kitagawa, "A Multiple Continuous Query Optimization Method Based on Query Execution Pattern Analysis," DASFAA 2004, LNCS 2973, pp. 443-456, 2003.

[16] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid. Exploiting Predicate-Window Semantics over Data Streams. *ACM SIGMOD Record*, Vol. 35, Issue 1. March, pp. 555-568, 2006.

[17] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid, "Scheduling for Shared Window Joins over Data Streams," In Proc. 29th VLDB Conf., pp. 297-308, 2003.

[18] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar, "State-Slice: New Paradigm of Multi-Query Optimization of Window-Based Stream Queries," In Proc. 32nd VLDB Conf., pp.619-630, 2006.

[19] S. Krishnamurthy, M.J. Franklin, J. M. Hellerstein, and G. Jacobson, "The Case for Precision Sharing," In Proc. 30th VLDB Conf., pp. 972-986, 2004.

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

변 창 우

1999년 서강대학교 컴퓨터학과(공학사) 2001년 서강대학교 컴퓨터학과(공학석사). 2007년 8월 서강대학교 컴퓨터공학과(공학박사). 2007년 9월~현재 인하공업전문대학 컴퓨터시스템과 전임강사. 관심분야는 XML 질의 처리기, 동적인 데이타베이스에서의 트랜잭션 관리, 역할기반 접근제어 모델, XML 접근제어, 프라이버시

이 헌 주

2005년 서강대학교 컴퓨터학과(공학사) 2007년 서강대학교 컴퓨터학과(공학석사) 2007년 2월~현재 (주)온더아이티 정보기술 연구소 대리. 관심분야는 다중 질의 최적화, 데이타 스트림 관리 시스템, XML 질의 처리, XML 데이타베이스

박 석

1978년 서울대학교 계산통계학과(이학사) 1980년 한국과학기술원 전산학과(공학석사). 1983년 한국과학기술원 전산학과(공학박사). 1983년 9월~현재 서강대학교 컴퓨터학과 교수. 1989년~1991년/2002년~2003년 University of Virginia 방문교수. 1997년 2월~현재 한국정보보호학회 이사. 2005년 한국정보과학회 부회장. 2004년 1월~2005년 한국정보과학회지 편집위원장. 2006년 KCC 2006 한국컴퓨터종합학술대회 프로그램위원장. 2006년 VLDB Panel Co-chair. 2007년 DASFAA Workshop Co-chair. 관심분야는 데이타베이스 보안, 실시간 시스템, 트랜잭션 관리, 데이타웨어하우스, 웹과 데이타베이스, XML, XML 데이타베이스 시스템에서의 필터링 기법, 역할기반 접근제어 임