엑스플러스 조인 : 조인 중복체크의 오버헤드를 줄이기 위한 개선된 방법

백주현° 박성욱 정성원
서강대학교 컴퓨터학과
{jhbaek7°, psw0405}@mclab.sogang.ac.kr, jungsung@sogang.ac.kr

X+ Join : The improved X join scheme for the duplicate check overhead reduction

Joo-hyun Baek° Sung-wook Park, Sungwon Jung
Dept. of Computer Science, Sogang University

요 약

유비쿼터스(Ubiquitous)환경과 같이 외부로부터 입력되는 데이터가 stream의 형식으로 실시간으로 들어오고, 입력의 끝을 알 수 없는 환경에서는 기존의 join방식으로는 문제를 해결 할 수 없다. 또한 이러한 환경 하에서는 데이터의 크기나 특성이 모두 다르고 네트워크 상태에 따라 입력이 많은 영향을 받게 된다. 이런 stream환경의 join연산을 위하여 double pipelined hash join, Xjoin, Pjoin등 많은 알고리즘이 기존의 연구를 대표하여 왔다. 그 중 Xjoin은 symmetric hash join과 hybrid hash join의 특징들을 이용해서 들어오는 data의 흐름에 따라서 reactive하게 join과정을 조절함으로써 streaming data에 대한 join을 수행한다. 그러나 여러 단계의 수행에 따른 연산의 중복결과를 체크하기 위한 overhead로 인해 성능이 떨어진다. 이 논문에서는 이러한 점을 개선하기 위해서 Xjoin의 수행과정을 수정한 방법을 제시할 것이다. 각 partition마다 구분자만을 추가함으로써 간단하게 중복을 만들어내지 않는 방법을 제안하고 불필요한 연산과 I/O를 줄일 수 있도록 partition선택방법을 추가할 것이다. 이를 통해서 중복된 연산인지 체크하는 과정을 상당히 단순화함으로써 좀 더 좋은 성능을 가지게 될 것이고 또한 timestamp를 저장해야 하는 overhead를 줄여서 전체 연산에 필요한 저장 공간을 절약할 수 있다.

1. 서 론

기존의 데이터베이스에서의 질의 연산은 table사이에서 tuple들을 단위로 연산 되어졌고, 영구적인 저장 공간 내에서 메모리 제약 없이 실행되어져 왔다. 그러나 요즘과 같이 유비쿼터스 시대에서는 고정된 table에서가 아닌 실시간 stream에서 질의 연산이 행하여진다. 따라서 stream의 끝을 알 수 없어 기존의 환경에서와 같이 카티션 프로덕트(cartesian product)를 실행하여 join결과를 생산해 낼 수 없다. 따라서 stream형태로 지속적인 입력을 받아 처리하고, stream형태의 출력을 데이터가 들어올 때 마다 바로 처리해서 결과를 출력하는 pipeline형식의 방식으로 처리하는 기술이 필요하다. 데이터베이스의 질의 처리 작업 중에서 가장 빈도 높은 연산이 바로 join이다. 전통적인 join 방식은 Nested Loop join, Hash join, Merge join등으로 기존의 데이터베이스 환경에서 효율적으로 작동하도록 고안되어져 왔다. 따라서 기존의 join알고리즘으로는 유비쿼터스 환경의 join연산에 많은 문제점을 야기 시킨다. 이러한 문제점들을 해결하기 위하여 stream환경에서의 join은 double pipelined hash join, Xjoin, Pjoin등 다양한 방식으로 연구되어져 왔다.

그러나 stream환경에서도 모든 join의 결과를 산출할 수 있는 방법이 필요하다. Streaming data는 특성상 random access가 불가능하고 순차적으로 들어오는 data를 처리해야 한다. join연산을 수행해서 모든 결과를 만들어 내기 위해서는 시간적으로 이전에 들어왔던 data와 나중에 들어온 data를 join해야 한다. 그래서 들어오는 data를 버릴 수 없고 저장해 두어야 하는데 streaming data는 data의 양이 memory에 전부 저장할 수 없을 정도로 많은 것이 일반적이다. 또한 data의 arrival rate가 일정하지 않을 수 있다[1, 2]. Streaming data의 source는 여러 가지가 될 수 있다. 예를 들면 internet에서 먼 곳에 있는 data를 받아와서 처리 할 수도 있으며 sensor network과 같은 환경에서는 각 sensor로부터 자료를 읽어 와야 할 수도 있다. 이러한 환경에서는 data가 어떻게 들어올 지 예측하기 어렵다. 들어오는 data가 거의 없다가 순간적으로 많아질 수도 있으며, 한쪽에서는 data가 들어오지만 다른 쪽에서는 data의 전송이 지연되어 늦게 들어올 수도 있다. 이러한 문제는 결과를 효과적으로 빠르게 내보내주기 어렵게 만드는 요인이 된다[4].

여기에서는 streaming data에 대한 join방법으로 제안된 X Join에 대해서 살펴보고 문제가 되는 부분을 수정한 알고리즘을 제안할 것이다. 먼저 2장에서는 X Join에 대해서 간단히 살펴보고 문제점을 지적할 것이다. 3장에서는 이것을 해결하기 위해서 수정된 알고리즘을 제안할 것이다. 마지막으로 4장에서는 이에 대한 결론을 내릴 것이다.

2. 관련 연구

본 논문에서 제안하는 기법을 설명하기 위해서 먼저 Xjoin에 대해서 설명할 것이다. 그리고 Xjoin의 문제점을 살펴보겠다. X Join은 기본적으로 symmetric hash join과 비슷하게 동작한다[3, 4, 5]. X Join은 각 입력의 tuple들을 hash를 이용해서 여러 개의 partition으로 나누어서 관리한다. 전체적인 수행은 세 단계로 나누어 수행된다. 첫 번째 단계는 입력이 정상적으로 들어올 때에 수행되는 단계이다. Tuple이 들어오면 그림 1과 같이 반대쪽에 hash를 이용해서 proving해서 결과를 출력하고 tuple이 들어가야 할 memory의 partition에 저장된다. 만약 memory가 꽉 차서 더 이상 넣을 수 없을 경우 가장 큰 partition을 선택해서 disk로 내보낸다[4].

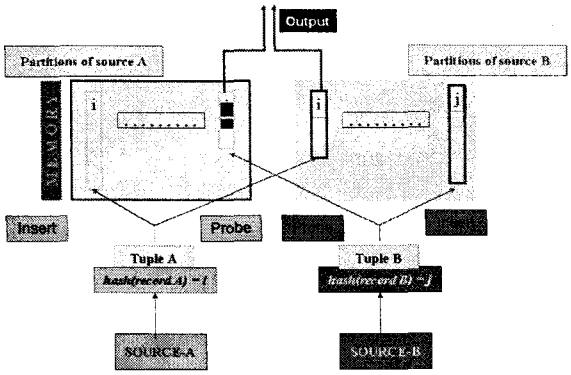

그림 1 Memory-to-Memory joins

두 번째 단계는 입력이 지연되거나 일시적으로 멈춰서 더 이상 들어오는 입력으로 새로운 결과를 만들어 낼 수 없을 경우에 수행된다. 이때에는 그림 2와 같이 disk에 있는 partition중에 하나를 선택해서 반대쪽 입력의 memory에 있는 partition과 join을 수행하게 된다[4].

세 번째 단계는 모든 입력이 끝나게 되면 지금까지 join이 일어나지 못한 pair들을 모두 계산해 주기 위한 단계이다. Disk에 있는 partition을 선택해서 반대쪽의 memory resident partition과 disk resident partition모두와 join을 수행한다[4]. 이 세 가지 단계를 통해서 X Join은 어느 한쪽의 입력이 지연되거나 일시적으로 멈추더라도 지속적으로 결과를 출력할 수 있고 memory overflow가 발생하더라도 대처할 수 있다. 그러나 기본적으로 join은 memory에 있는 tuple들끼리만 수행되기 때문에 Disk로 내보내진 tuple들은 join이 부분적으로 수행되지 못할 수 있다. 이 때문에 다시 memory로 읽어 와서 join을 수행해야 하는 과정이 필요한데 이 과정에서 많은 I/O overhead가 발생하게 된다. 또한 중복된 결과를 만들어 낼 수 있기 때문에 이전에 join을 수행한 pair인지 구분이 되어야 한다. 만약 중복된 결과를 만들어내는 tuple을 disk에서 읽어왔다면 불필요한 I/O overhead가 발생한 것이다.

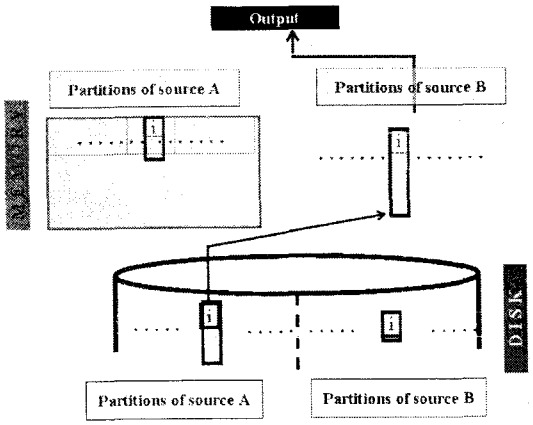

그림 2 Disk-to-Memory joins

또한 이전에 join했던 pair인지 검사하기 위해서 X Join에서는 중복을 구분해내기 위해서 timestamp를 이용한다[4]. 각 tuple마다 ATS(arrival time stamp), DTS(departure time stamp)를 저장한다. ATS와 DTS는 각각 source로부터 tuple이 처음 도착한 시간과 tuple이 memory로부터 처음 disk로 flush된 시간을 나타낸다. 따라서 DTS와 ATS의 차이는 해당 tuple이 memory에 있었던 기간을 의미하는데 그림 3과 같이 이 기간이 겹치는 tuple의 pair는 첫 번째 단계에서 join했던 pair이다.

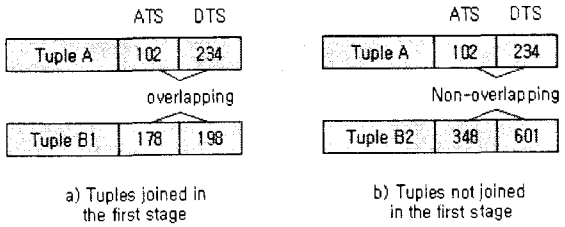

그림 3 첫 번째 단계에서 join된 tuple들을 검사

두 번째 단계에서 join되었던 pair인지 구분하기 위해서는 그림 4와 같이 DTSlast와 ProbeTS 값들을 두 번째 단계가 일어날 때마다 linked list형태로 가지고 있다가 join하려는 tuple의 ATS, DTS 범위와 겹치는 지를 확인한다.

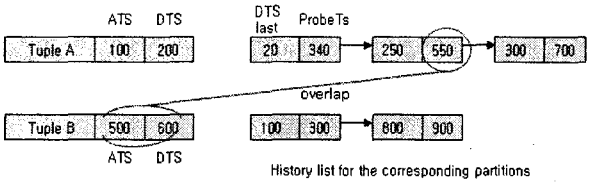

그림 4 두 번째 단계에서 join된 tuple들을 검사

이런 값들을 유지하고 있는 것은 상당한 memory overhead이다. 연산해야 할 tuple이 상당히 많다면 timestamp를 저장하기 위해서 필요한 공간도 무시할 수 없을 것이다. 중복을 검사하는 과정도 매번 수행되어야 하며 복잡하기 때문에 성능을 저하시키는 요인이 될 것이다.

3. Modefied X join

이번 장에서는 앞서 살펴본 문제들을 줄여서 X Join이 좀 더 효율적으로 동작할 수 있도록 하기 위해서 수정된 알고리즘을 제안한다.

첫 번째 단계는 기존의 X Join과 같은 방식으로 동작한다. 다만 overflow가 발생했을 때에는 overflow가 발생한 시점에 입력이 들어온 쪽의 memory resident partition 중에서 하나를 골라서 disk로 내보내는 것이 아니라 그림 5와 같이 partition 하나를 골라서 같은 hash 값을 가지는 양쪽의 memory resident partition모두를 disk로 내보낸다. 이 때에 모든 입력은 같은 수의 partition으로 나누어진다고 가정한다. 이것은 두 번째와 세 번째 단계에서의 중복처리를 간편하게 하기 위해서이다.

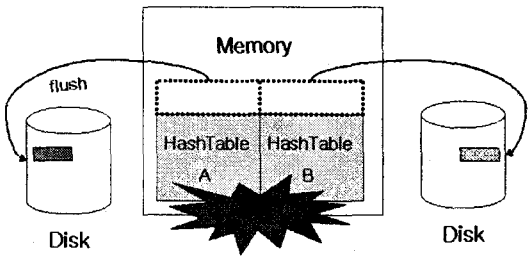

그림 5 overflow 발생시 momory 내용을 디스크로 저장하는 방법

이렇게 했을 때 발생할 수 있는 문제점은 모든 입력에서의 partition을 memory로 내보내기 때문에 해당 partition의 자료가 새로 들어왔을 때 memory에서 join을 수행할 수 있는 기회를 막아버리게 되는 것이다. 기존의 방법대로 한 쪽의 partition만을 disk로 내보내면 내보낸 쪽에서 해당 partition으로 tuple이 들어왔을 때 반대쪽에 남아있는 partition과 join을 수행할 수 있는 가능성이 있다. 하지만 모두 내보내게 되면 어느 한쪽 입력으로 해당 partition에 tuple이 들어오더라도 아직 다른 입력으로 해당 partition에 tuple이 들어오지 않았다면 join을 수행하지 못한다. 이러한 문제 때문에 어떤 partition을 선택해서 disk로 내보낼 것인지 선택하는 문제가 성능에 영향을 주는 중요한 문제가 된다. 선택하는 방법에는 표 1과 같이 여러 가지가 있을 수 있는데 partition의 총 size가 가장 작은 것을 선택하거나, partition의 총 size가 가장 큰 것을 선택할 수 있다. 가장 작은 것을 선택했을 경우에는 위에서 언급한 문제는 적을 수 있지만 충분한 공간을 확보하지 못할 수 있고 overflow가 자주 발생할 수 있다. 총 size가 가장 큰 것을 선택할 경우에는 위에서 언급한 문제가 발생할 가능성이 크다. 적절하게 선택할 수 있는 한 가지 방법으로 tuple들이 최근 일정기간 동안 해당 partition으로 들어오는 비율을 보고 모든 입력에서 이 비율의 합이 가장 적은 partition을 disk로 내보내는 방법이 있다. 해당 partition으로 들어오는 비율이 작으면 문제가 발생할 가능성이 적다고 볼 수 있다. 혹은 각각의 입력에 대해서 해당 partition으로 들어오는 비율을 구해서 차가 가장 작은 tuple을 선택하는 방법도 있다. 양쪽에서 입력이 골고루 들어오는 경우에는 양쪽을 빼더라도 영향을 적게 받을 것이다.

표 1 disk로 보내질 partition 선택 방법

◎ partition 선택조건
1. 가장 작은 partition
2. 가장 큰 partition
3. input rate가 가장 작은 partition
4. input rate의 차이가 가장 작은 partition

두 번째 단계에서는 disk에서 읽어올 partition을 고르는 방법을 추가한다. 기존의 X Join에서는 두 번째 단계에서 읽어올 partition을 랜덤방식으로 선택한다. 랜덤으로 선택하는 경우 불필요하게 중복된 join을 수행할 수 있다. X Join에서는 두 번째 단계를 연속적으로 수행할 수 있는데 이때에 random방식으로 같은 partition을 여러 번 선택한다면 불필요한 연산을 수행하게 되는 것이다. 이런 문제를 방지하기 위해서 memory의 각 partition에 update가 일어났는지를 check하는 bit를 유지한다. 이 bit는 partition에 새로운 tuple이 들어오면 1로 setting되고 두 번째 단계에서 partition이 선택되거나 해당 partition이disk로 내보내

질 경우에 0으로 setting된다. 두 번째 단계에서 disk로부터 읽어올 partition을 선택할 때에 다른 입력 쪽의 같은 hash 값을 가지는 memory resident partition의 check bit가 1인 partition중에서만 선택한다. 즉, memory에 새로운 내용이 있어서 join이 새로운 결과를 만들어 낼 수 있는 것들만 선택한다. 이 방법을 이용해서 두 번째 단계가 연속적으로 수행될 때 같은 partition이 여러 번 선택되어 불필요한 join을 수행하는 것을 피하고 memory에 새로운 내용이 있는 경우에만 해당 partition을 선택하게 함으로써 결과를 낼 수 있는 partition만이 선택되도록 한다.

```
select_partition_number()
{
  for(;;)
  {
    x = generate a random number
          (0 ~ the number of partition)
    if(check bit of xth partition == 1)
       break;
  }
  return x
}
```

그림 6 Disk에서 partition 고르는 알고리즘

두 번째 단계가 수행될 때 memory에는 같은 tuple이 계속 있을 수도 있다. 이 경우 해당 partition이 두 번째 단계에서 여러 번 수행될 경우 중복된 연산을 수행될 것이다. 이것을 피하기 위해서 두 번째 단계가 수행되고 난 직후에 수행된 disk resident partition과 memory resident partition에 구분자 S를 partition의 맨 뒤에 추가한다. 두 번째 단계를 수행할 때에 그림 7과 같이 역순으로 tuple을 읽으면서 수행한다. Memory에 있는 partition도 역순으로 읽어서 probe하는데 disk에서 구분자 S를 읽게 되면 구분자 S 다음에 있는 tuple들은 이전에 한번 join했던 tuple들이라는 것을 의미한다. 마찬가지로 Disk partition에서 구분자 S를 읽게 되면 이후의 tuple들은 memory partition에서의 구분자 S 이후의 tuple들과는 join을 수행하지 않는다. 또한 세 번째 단계에서 이용하기 위해서 두 번째 단계를 수행한 memory partition의 tuple들에는 구분자 S에 들어간 값과 같은 값으로 flag를 저장한다.

세 번째 단계에서는 지금까지 join이 이루어지지 못한 pair들에 대해서 join을 수행해야 한다. 여기에서는 disk resident partition끼리 join을 수행하게 된다. 이때에도 이전에 join이 수행됐던 tuple에 대해서 중복 수행되지 않도록 하기 위해서 좀 더 복잡한 방법이 추가되어야 한다.

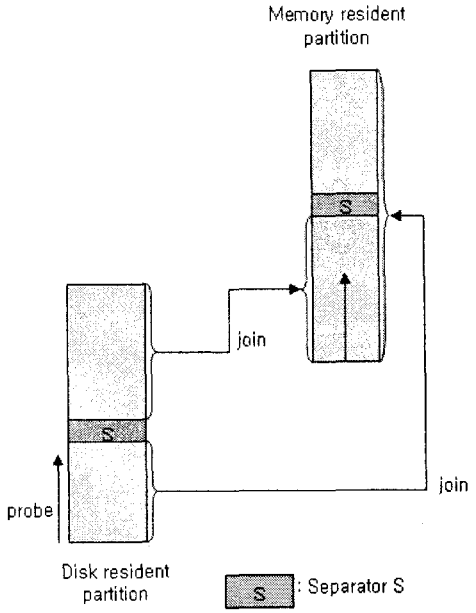

그림 7 separator S 추가를 이용한 probe

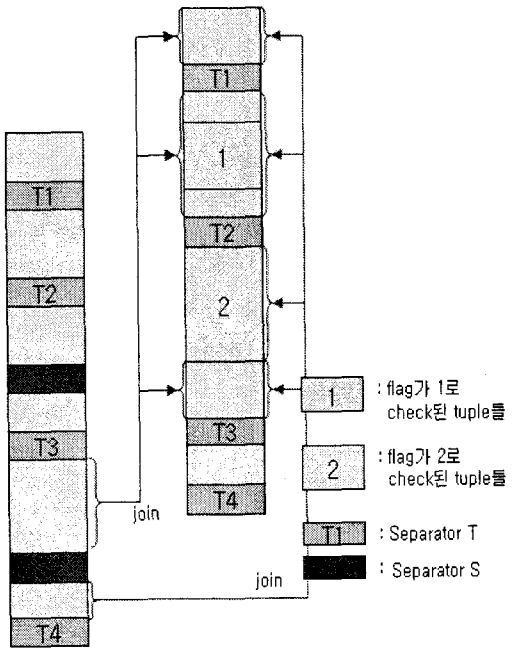

그림 8 separator T를 이용한 join

Memory에 있던 tuple이 disk로 내보내질 때에는 맨 뒷

부분에 counter나 timestamp를 가지는 구분자 T를 그림 8과 같이 추가한다. 양쪽 입력에서 동시에 disk로 내보내지므로 이 구분자 T들로 같은 시간 동안 memory에 있었던 tuple들을 구분해 낼 수 있다. 또한 두 번째 단계에서 join했던 것을 구분해 내기 위해서 두 번째 단계에서 추가한 구분자 S와 tuple에 저장된 flag를 이용한다. 세 번째 단계에서 join을 할 때에도 역순으로 읽어서 수행하며, 구분자 T를 이용해서 같은 시간 동안 memory에 있지 않았던 구역을 알아낸다. 그 구역의 tuple들 중에서 구분자 S와 flag을 이용하여 flag값이 구분자 S의 값보다 작은 tuple들을 선택해서 join을 수행한다.

4. 결 론

기존에 제안되었던 X Join 기법은 streaming data환경에서 모든 가능한 join결과를 만들어 내기 위한 방법으로 제안되었다. streaming data환경에서는 모든 data를 memory에 저장할 수 없기 때문에 secondary storage를 이용하는 것이 필연적이다. X Join은 이러한 환경에서 secondary storage를 이용해서 join을 수행하는 기법을 제안하였지만 앞에서 살펴본 바와 같이 연산을 수행하기 위해 필요한 overhead가 크다는 문제점이 있었다.

이 논문에서는 이러한 문제점을 제안하기 위해 중복된 join연산을 피하는 방법을 제안하였다. X Join에서는 중복된 join연산인지 모든 연산을 수행하기 전에 check를 해야 하며 이를 위해서 각 tuple마다 timestamp를 유지해야 하고, 모든 partition마다 linked list를 유지해야 한다. 여기에서는 이러한 방법대신에 기존에 연산을 수행했던 부분을 구분할 수 있는 구분자를 필요한 곳에 끼워 넣어 둠으로써 매번 check를 하지 않고서도 중복된 연산을 피할 수 있다. 또한 이 과정을 수행하기 전에 연산 결과를 만들어 낼 수 있는 partition만 선택적으로 수행하는 방법을 추가함으로써 X Join에서 발생할 수 있는 필요 없는 연산을 줄일 수 있다.

이 논문에서 제안한 방법에서는 X Join과 달리 memory resident partition을 disk로 빼낼 경우에 양쪽 입력 모두에서 빼내는 방법을 취한다. 이러한 경우에 조인 연산이 가능한 tuple들을 memory에서 제거함으로써 생길 수 있는 연산 성능저하에 대해서 좀 더 연구해 볼 필요가 있다. 또한 성능저하가 발생할 경우에 이 논문에서 제안한 대처법 이외에도 상황에 따라 좀 더 효율적으로 대처할 수 있는 방법도 필요할 것이다.

이 외에도 좀 더 성능을 개선하기 위해서 disk I/O의 overhead를 줄일 수 있는 방법을 좀 더 연구해 볼 필요가 있다. disk I/O overhead를 줄이기 위해서는 disk를 access하는 횟수를 줄이거나 혹은 disk를 access하는 전체적인 양을 줄일 수 있는 기법에 대한 연구가 필요할 것이다.

참 고 문 헌

[1] L. Amsaleg, M. J. .Franklin, A. Tomasic, and T. Urhan. "Scrambling Query Plans to Cope With Unexpected Delays." *PDIS Conf.*, Miami, USA, 1996.

[2] L. Amsaleg, M. J. .Franklin, and A. Tomasic. "Dynamic Query Operator Scheduling for Wide-Area Remote Access." *Journal of Distributed and Parallel Databases*,Vol. 6, No. 3, July 1998.

[3] W. Hong, M. Stonebraker. "Optimization of Parallel Query Execution Plans in XPRS." *Distributed and Parallel Databases*, 1(1):9-32, 1993.

[4] T. Urhan, M. J. .Franklin. "XJoin: Getting Fast Answers from Slow and Bursty Networks." *University of Maryland Technical Report, CS-TR-3994.*,February, 1999.

[5] A. N. Wilschut, and P. M. G. Apers. "Dataflow Query Execution in a Parallel Main-Memory Environment." *1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, 1991.