동적 프로세서 할당 기법을 이용한 파이프라인 해쉬 결합 알고리즘

(A Pipelined Hash Join Algorithm using Dynamic Processor Allocation)

원 영 선 † 이 동 련 †† 이 규 옥 ††† 홍 만 표 ††††

(Youngsun Weon) (Dongryun Lee) (Kyu-Ock Lee) (Manpyo Hong)

요 약 본 논문에서는 부쉬 트리를 할당 트리로 변환한 후 결합 연산을 수행하면서 실제 실행시간을 동적으로 계산하고 그 결과에 의해 실시간에 프로세서를 할당하는 동적 프로세서 할당 기법을 이용한 파이프라인 해쉬 결합 알고리즘을 제안하였다. 프로세서를 할당하는 과정에서 초기 릴레이션의 기본 정보만을 이용하여 미리 프로세서를 할당하는 기존의 정적 프로세서 할당 기법은 정확한 실행시간을 예측할 수 없었다. 따라서 본 논문에서는 할당 트리 각 노드의 실행결과를 포함한 결합 과정 중의 정보를 다음 노드의 실행시간에 충분히 반영하는 동적 프로세서 할당 기법을 제안하였으며, 이로써 프로세서를 효율적으로 분배하고 전체적인 실행시간을 최소화하였다.

또한 전체적인 질의 실행시간을 줄이기 위하여 결합 가능성이 없는 튜플들을 제거한 후 결합 연산을 수행할 수 있도록 해쉬 필터 기법을 이용하였다. 결합 연산을 수행하기에 앞서 모든 결합 속성값에 대해 해쉬 필터를 생성하는 정적 필터 기법은 모든 결합 연산의 중간 결과로 발생할 수 있으나 최종 결과 릴레이션의 튜플이 될 수 없는 튜플들까지도 모두 추출이 가능하다. 따라서 각각의 결합 연산 직전에 해쉬 필터를 생성하는 동적 필터 기법에 비해 결합 가능성이 없는 튜플을 최대한 제거할 수 있으며 이로써 결합 연산의 실행비용을 크게 줄일 수 있었다.

Abstract In this paper, we proposes a pipelined hash join algorithm utilizing the dynamic processor allocation scheme that allocates processors according to the execution time expected. The execution time of a node in the allocation tree converted from a bushy tree is estimated from the size of the relations generated prior to executing the node. As the processors are allocated by the basic information based on the size of the initial relations under the conventional static processor allocation scheme, the execution time for each node cannot be measured exactly. Under a new dynamic processor allocation scheme proposed in this paper, the progress information of each node at the allocation tree and the information during the join operation are sufficiently reflected on the execution time of the following node. This new scheme achieves a more effective allocation of processors and minimizes the overall execution time as well.

Furthermore, in order to save the execution time for a query, the new technique employs a hash filter by which the join operation can be performed after removal of the tuples with no join operation possibility. While the dynamic filter generates the hash filter immediately before each individual join operation, the static filter generates the hash filter for all the join attribute values before starting the join operation. As only those tuples that may be able to join are extracted in advance, the size of the initial relations is minimized. In so doing, we can save the execution cost of the join operation to a great extent.

· 이 논문은 1999년도 두뇌한국21사업에 의하여 지원되었음

† 비 회 원 : 아주대학교 컴퓨터공학과
ysweon@madang.ajou.ac.kr
†† 비 회 원 : 아주대학교 정보통신전문대학원
ryuni@madang.ajou.ac.kr
††† 정 회 원 : 한국기계연구원 연구원
kolee@kimm.re.kr
†††† 종신회원 : 아주대학교 정보통신전문대학원 교수
mphong@madang.ajou.ac.kr

논문접수 : 2000년 1월 18일
심사완료 : 2000년 11월 23일

1. 서 론

효율적인 자료 처리를 위하여 빠른 실행시간을 요구하는 대용량 데이타베이스 시스템에서는 병렬화를 통하여 성능 향상을 기대할 수 있다. 특히 데이타베이스 시스템에서 다른 연산에 비해 비교적 많은 실행비용을 요구하는 결합 연산은 데이타베이스의 규모가 커지고 질의가 점점 복잡해짐에 따라 그 비용은 더욱 증가하게 된다. 따라서 복잡한 다중 결합 질의의 효율적 실행은 데이타베이스 시스템의 전체 성능에 커다란 영향을 미친다[1,2,4,5].

이와 같은 복잡한 다중 결합 질의를 효율적으로 처리하기 위하여 병렬성을 이용한 많은 연구가 제안되었다. 병렬성은 하나의 결합 연산에 대해서 다수의 프로세서들이 병렬로 작업을 수행하는 연산자내 병렬성(intra-operator parallelism)과 다중 질의의 병렬 수행을 위한 연산자간 병렬성(inter-operator parallelism)으로 나눌 수 있다.

일반적으로 질의 계획(query plan)은 결합 순차 트리(join sequence tree)라 불리는 연산 트리로 변환된다. 여기서 말단 노드는 입력 릴레이션을, 중간 노드는 두 자식 노드에 할당된 두 릴레이션의 결합으로부터 생성된 결과 릴레이션을 나타낸다. 연산 트리는 그 형태에 따라 좌향(left-deep) 트리, 우향(right-deep) 트리 그리고 부쉬(bushy) 트리로 분류되며, 좌향 트리와 우향 트리를 선형 실행 트리(linear execution trees) 또는 순차 결합 순서(sequential join sequences) 라고 부른다.

또한 다양한 결합 방법 중에서 해쉬 결합은 파이프라인 기회를 제공하고 다른 방법에 비해 우수한 성능을 보이고 있다[2,7,8]. 해쉬 결합의 파이프라인은 여러 단계로 이루어져 있으며, 각각의 단계는 여러 프로세서에 의해서 병렬적으로 실행될 수 있는 하나의 결합 연산으로 이루어져 있다.

최근까지 주로 선형 실행 트리에 대한 연구가 진행된 반면, 최근들어 하드웨어의 급속한 발전과 상당히 복잡해진 질의로 인해 부쉬 트리에 대한 연구가 이루어지고 있다. 또한 연산자간 병렬화와 연산자내 병렬화의 통합적 접근 및 결합 순서 스케줄링과 프로세서 할당을 다루는 연구들이 제안되었다[4]. 그 중 다중 질의 처리를 위한 효율적인 방법으로 할당 트리를 이용한 정적 프로세서 할당 방법이 제안되었다[1]. 이 방법은 파이프라인이 가능한 부쉬 트리의 노드들을 그룹핑하여 할당 트리를 생성하고 기본 릴레이션의 초기 정보를 이용하여 상향식(bottom-up) 방법으로 누적 실행비용을 계산 한 후 다시 하향식(top-down) 방법에 의해 프로세서를 할당한다. 그러나 단지 기본 릴레이션의 초기 정보만을 이용하여 실행시간을 예측하고 프로세서를 할당함으로써 정확한 실행시간 예측이 불가능하다. 또한 기본 릴레이션의 초기 정보를 이용하여 누적 실행비용을 계산함으로써 결합 연산을 수행하기 전에 모든 프로세서 할당이 이루어지는 정적 프로세서 할당 기법으로 인해 전체 질의어의 실행시간을 최소화 할 수 있는 프로세서 할당 방법을 찾는데는 한계가 있다.

따라서 본 논문에서는 병렬화의 최대 효과를 이루기 위하여 부쉬 트리를 할당 트리로 변환한 후 실행시간을 동적으로 계산하고 그 결과에 의해 프로세서를 할당하는 동적 프로세서 할당 기법을 제안한다. 또한 전체적인 질의 실행시간을 줄이기 위하여 결합 가능성이 없는 튜플들을 제거한 후 결합 연산을 수행할 수 있도록 해쉬 필터 기법을 이용한다.

제안된 동적 프로세서 할당 방법의 성능을 평가하기 위하여 비용 모델을 세우고 기존의 효율적 방법으로 알려진 할당 트리를 이용한 정적 프로세서 할당 방법과 비교하였다.

본 논문은 다음과 같이 구성되어 있다.

2장에서는 관련 연구로써 할당 트리를 이용한 정적 프로세서 할당 방법에 대해 살펴본다. 본 논문에서 새롭게 제안한 파이프라인 해쉬 결합을 기반으로 하는 정적 프로세서 할당 방법과 질의 실행시간의 개선을 위한 해쉬 필터 적용 방법은 3장에서 기술하고, 4장에서는 새로운 동적 프로세서 할당 방법의 성능을 평가하기 위한 분석 모형을 제시한다. 5장에서는 분석 모형을 통해 다양한 방법으로 성능을 평가하며, 정적 프로세서 할당 방법과 성능 비교를 한다. 마지막으로 6장에서는 본 논문에서 얻은 결과를 정리한다.

2. 할당 트리를 이용한 정적 프로세서 할당 기법

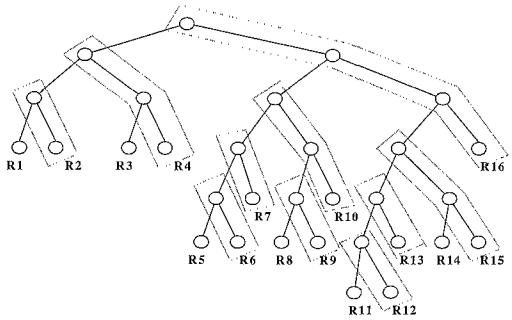
부쉬 트리 형태로 표현된 다중 결합 질의를 처리하기 위한 방법으로 부쉬 트리에서 파이프라인이 가능한 노드들을 그룹핑하여 생성된 할당 트리를 이용하는 방법이 제안되었다[1]. 이 방법은 부쉬 트리를 각 노드가 하나의 파이프라인으로 표현되는 할당 트리로 변환한 후, 상향식 방법에 의해 누적 실행비용을 계산하고 다시 하향식 방법에 의해 프로세서의 수를 결정한다. 이 방법은 하나의 부(parent) 노드의 실행 전까지 모든 자식(child) 노드들이 거의 동시에 실행을 마칠 수 있도록 동기 실행시간 개념을 이용하며, 결합 연산을 수행하기 전에 단

지 기본 릴레이션들의 카디널리티와 속성값이 가질 수 있는 도메인의 카디널리티 만을 고려하여 누적 실행비용을 계산하고, 그 값에 의해 프로세서를 할당하는 정적 프로세서 할당 기법을 이용한다.

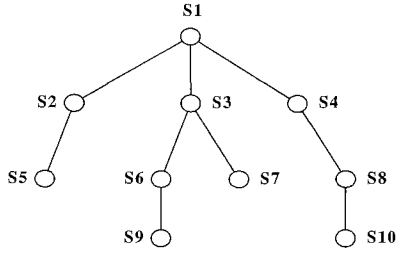
다음은 부쉬 트리 형태로 표현된 다중 결합 질의를 할당 트리의 형태로 변환하고 정적 프로세서 할당 기법에 의해 프로세서를 할당하는 방법에 대해 살펴본다.

2.1 할당 트리

다중 해쉬 결합 질의를 파이프라인 방식으로 처리하기 위한 방법으로 부쉬 트리에서 파이프라인이 가능한 노드들을 그룹핑하여 새로운 형태의 할당 트리를 생성하는 방법이 제안되었다. 이 방법은 먼저 부쉬 트리로 표현된 질의에서 파이프라인이 가능한 노드들을 찾는다. 다음 파이프라인이 가능한 노드들을 그룹핑하여 할당 트리의 한 노드로 변환한다. 그림 1은 부쉬 트리에서 파이프라인을 식별하여 할당 트리로 변환하는 과정이다. (a)의 각각의 파이프라인의 그룹들은 (b)의 한 노드로 변환됨을 알 수 있다.



(a) 부쉬 트리에서 파이프라인을 식별한 결과



(b) 할당 트리

그림 1 부쉬 트리를 할당 트리로 변환

2.2 정적 프로세서 할당 기법

파이프라인이 가능한 결합 연산을 그룹핑하여 할당 트리로 변환이 이루어지면 각각의 할당 트리 노드에 필요한 프로세서를 할당한다. 먼저 기본 릴레이션들의 카디널리티와 속성값이 가질 수 있는 도메인의 카디널리티 만을 고려하여 할당 트리 각 노드의 실행비용을 계산한다. 다음 할당 트리 각 노드의 실행비용을 상향식 방법에 의해 누적하여 각 노드의 누적 실행비용을 계산한다. 각 노드의 누적 실행비용이 계산되면 루트 노드로부터 하향식 방법에 의해 프로세서의 수를 결정한다. 이때 부 노드를 실행하기 위해서는 반드시 자식 노드들의 실행이 완료되어야 하므로 자식 노드들이 거의 동시에 실행을 마칠 수 있도록 동기 실행시간 개념을 이용하여 프로세서를 할당한다.

따라서 할당 트리 각 노드의 누적 실행비용의 계산이 이루어지면 각 노드에서 필요한 프로세서가 수가 결정되고 결합 연산을 수행하기 전에 모든 프로세서 할당이 이루어진다. 이러한 정적 프로세서 할당 방법의 실행과정은 다음과 같다.

step 1: 하나의 결합 순서 휴리스틱을 이용하여 부쉬 실행 트리를 결정한다.

step 2: 부쉬 트리로부터 파이프라인이 가능한 릴레이션들을 그룹핑하여 할당 트리로 변환한다.

step 3: 상향식 방법에 의해 할당 트리 각 노드의 누적 실행비용을 계산한다.

step 4: 하향식 방법에 의해 할당 트리 각 노드에 프로세서를 할당한다.

3. 동적 프로세서 할당 기법을 이용한 파이프라인 해쉬 결합 알고리즘

다중 질의 해쉬 결합 알고리즘을 효율적으로 수행하기 위해서는 병렬 실행의 효과를 최대화하여야 한다. 이는 프로세서의 유휴 시간을 최소화함으로써 주어진 프로세서를 최대한 활용함을 의미한다.

다중 질의 해쉬 결합 알고리즘의 효율적 수행을 위한 방법으로 알려진 할당 트리를 이용한 정적 프로세서 할당 기법은 누적 실행비용을 계산함에 있어서 단순히 기본 릴레이션들의 카디널리티와 속성값이 가질 수 있는 도메인의 카디널리티 만을 고려하였다. 이와 같은 정적 프로세서 할당 방법은 결합 연산을 수행하기에 앞서 프로세서의 할당이 모두 이루어지므로 각각의 결합 연산의 결과로 생성되는 중간 릴레이션들의 크기를 전혀 고려하지 않음으로서 정확한 실행시간 예측이 어렵다. 따라서 보다 정확한 실행시간 예측을 위하여 동적으로 중간 릴레이션의 정보를 이용하여 실제 실행시간을 예측하고 프로세서 할당을 위한 정보로 활용함으로써, 유휴 시간을 최소로 하며 동시에 전체 실행시간을 줄일 수 있는 동적 프로세서 할당 기법을 제안한다.

또한 결합 연산을 수행하기에 앞서 결합 가능성이 없는 튜플들을 미리 제거함으로써 파이프라인 해쉬 결합의 결합률을 높이고, 보다 효율적인 프로세서 할당을 통해 전체 실행시간을 단축할 수 있는 해쉬 필터 기법을 적용한다.

3.1 동적 프로세서 할당 기법

정적 프로세서 할당 기법은 결합 연산을 수행하기에 앞서 기본 릴레이션의 정보만을 이용하여 누적 실행비용을 계산하고, 그 값에 의해 할당 트리의 모든 노드에 각각 프로세서를 할당한다. 그러나 기본 릴레이션의 기본 정보에 의해 누적 실행비용을 계산할 경우 각 결합 연산의 결합률에 따라 오차가 발생하고 할당 트리의 결합 연산이 진행될수록 그 오차는 더욱 증가하게 되며 정확한 비용 계산이 불가능하게 된다. 따라서 기본 릴레이션의 정보에 의해 누적 실행비용을 계산함으로써 발생하는 이와 같은 오차를, 결합 연산을 수행하는 과정에서 중간 릴레이션의 정보를 이용하고 동시에 할당 트리 각 노드의 결합 연산 실행시간을 계산함으로써 정확한 프로세서 할당 정보로 이용할 수 있다. 또한 기존의 정적 프로세서 할당 기법이 동기 실행시간 개념을 이용해 한 노드의 모든 자식 노드들이 가능한 동시에 끝마칠 수 있도록 하였던 과정을, 병렬 실행이 가능한 할당 트리의 동일한 깊이의 모든 노드들이 가능한 동시에 끝마칠 수 있도록 함으로써 트리 실행의 병렬도를 높일 수 있다.

정적 프로세서 할당 기법이 상향식 방법에 의해 누적 실행비용을 계산하고 하향식에 의해 프로세서의 수를 결정하는 것에 비해, 동적 프로세서 할당 기법은 할당 트리의 각 깊이 별로 프로세서 할당과 결합 연산을 반복한다. 변환된 할당 트리를 $T_{allocation}$이라 하고, $T_{allocation}$의 각 노드를 S_x라 할 때, $T_{allocation}$ 각 노드들의 실행시간을 $Cost_{node}(S_x)$라 하자. 여기서 $T_{allocation}$ 각 노드의 실행시간 $Cost_{node}(S_x)$는 4장의 비용 모델로부터 얻을 수 있다. 또한 S_x에 할당된 프로세서의 수를 $\#P(S_x)$라고 할 때, 할당 트리 $T_{allocation}$ 각 노드에 할당된 프로세서 수는 다음과 같다.

$$\#P(S_x) = \lceil n_p \times \frac{Cost_{node}(S_x)}{\sum_{x \in equal_{level}} Cost_{node}(S_x)} \rceil$$

여기서, n_p는 프로세서의 수를, $equal_{level}$은 할당 트리에서 깊이가 동일한 노드들의 집합을 나타낸다.

즉, 할당 트리의 동일한 깊이에 있는 노드들의 실행시간을 계산한 후 각 노드들의 실행시간에 비례하여 모든 프로세서들을 할당한다. 따라서 할당 트리의 동일한 깊이의 노드들의 결합 연산 작업을 수행하기 위해 모든 노드들이 사용됨으로써 트리 실행의 병렬도를 최대한 살릴 수 있다. 이러한 동적 프로세서 할당 방법의 실행 과정은 다음과 같다.

step 1: 하나의 결합 순서 휴리스틱을 이용하여 부쉬 실행 트리를 결정한다.

step 2: 부쉬 트리로 부터 파이프라인이 가능한 릴레이션들을 그룹핑하여 할당 트리로 변환한다.

step 3: 상향식 방법에 의해 할당 트리의 깊이 단위로 다음의 과정을 반복한다.

1) 할당 트리 해당 깊이의 노드들의 실행비용을 계산한다.
2) 할당 트리 해당 깊이의 각 노드에 프로세서를 할당한다.

다음은 프로세서의 수가 32개 일 때 그림 1의 예에 대한 동적 프로세서 할당 과정을 나타낸다. 먼저 그림 1과 같이 부쉬 트리를 파이프라인이 가능한 노드들의 그룹핑을 통해 할당 트리로 변환한다. 다음 상향식 방법에 의해 할당 트리 해당 깊이 각 노드의 실행시간을 계산하고, 각 노드의 실행시간에 의해 프로세서를 할당한다. 그림 2는 각 노드의 실행비용을 4장의 비용 모델에 의해 계산한 후 그 비율에 의해 프로세서를 동적으로 할당하는 과정을 보이고 있다.

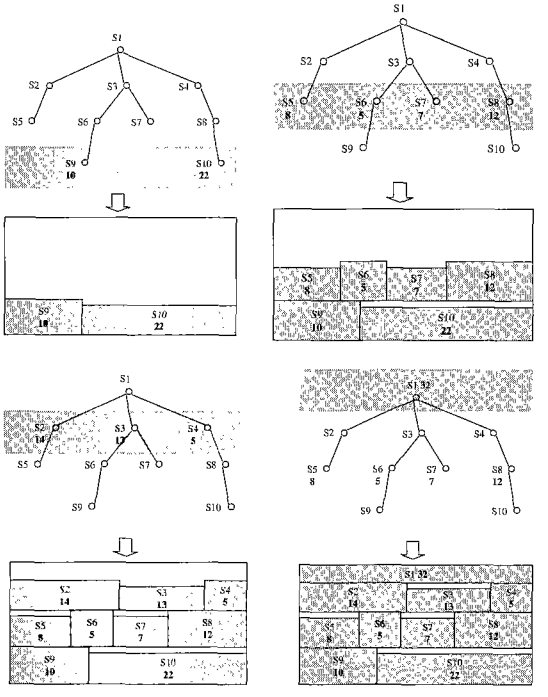


그림 2 동적 프로세서 할당 기법

DynamicProcessorAllocation

for all S_x *in* $equal_{level}$ {

$$\#P(S_x) = \lceil n_p \times \frac{Cost_{node}(S_x)}{\sum_{x \in equal_{level}} Cost_{node}(S_x)} \rceil$$

} *end for*

각 프로세서의 실행시간을 최대한 동일하게 적용함으로서 전체적인 병렬 실행시간을 최소로 할 수 있는 효율적인 동적 프로세서 할당 기법 모듈은 다음과 같다. 이 모듈은 할당 트리의 동일한 깊이에 대한 프로세서 할당 방법으로, 할당 트리의 전체 깊이 만큼 결합 연산과 함께 반복 수행된다.

3.2 해쉬 필터를 이용한 할당 트리의 실행

결합 연산을 수행하기에 앞서 결합 가능성이 없는 튜플들을 미리 제거한다면 결합 연산의 결합률을 높이고 전체적인 실행시간을 단축할 수 있다. 따라서 릴레이션의 결합 가능성을 확인하고 결합 가능성이 없는 튜플들을 릴레이션에서 제거하기 위해 해쉬 필터 기법을 이용한다. 이러한 해쉬 필터 생성 적용 방법으로는, 가장 간단한 방법인 결합 연산이 수행되기 직전 해당 속성값에 의해 해쉬 필터를 생성하고 적용하는 동적 필터(dynamic filter)와, 초기 단계에서 즉 결합 연산을 수행하기 전에 모든 결합 속성값에 대해 각각의 릴레이션에서 모두 적용하는 정적 필터(static filter)를 적용한다.

3.2.1 동적 필터(dynamic filter)

각각의 결합 연산을 수행하기에 앞서 해당 속성값에 대한 해쉬 필터를 생성하고 적용하는 방법으로 해쉬 필터의 생성 적용 비용이 정적 필터에 비해 적어 효율적이다. 그러나 초기 릴레이션의 크기를 충분히 줄이지 못하므로 결합 연산 과정에서 축소 가능 릴레이션에 대한 최대한의 효율을 가질 수 없으므로 결합률 측면에서의 효율은 정적 필터에 비해 떨어진다.

할당 트리의 각 노드는 우향 트리로 형성되며 우향 트리의 각 노드는 하나의 결합 연산을 의미한다. 이때 우향 트리의 각 노드에서는 결합 연산을 수행하기에 앞서 해쉬 필터를 생성하고 적용한다. 따라서 우향 트리의 한 노드에서 이루어지는 해쉬 필터 생성 및 적용 방법을 살펴보면 다음과 같다.

먼저 두 릴레이션 중 좌측 릴레이션의 튜플을 이용하여 해쉬 필터를 생성한다. 다음 우측 릴레이션의 튜플들을 해쉬 필터에 적용한다. 그림 3은 릴레이션 R과 릴레이션 S에 대해 속성 B에 대한 해쉬 필터 생성 적용 과정이다.

여기서 $HF(B)$는 릴레이션 R의 속성값 B에 의해 구

R

attr.A	attr.B
a1	b1
a2	b3
a3	b6
a4	b3

HF(B)

	B
0	1
1	1
2	0

S

attr.B	attr.C
b0	c1
b1	c2
b6	c6
b9	v7

New S

attr.*B*	attr.*B*
b0	b0
b1	b1
b2	b2
b5	b5
b6	b6
b8	b8
b9	b9
b11	b11

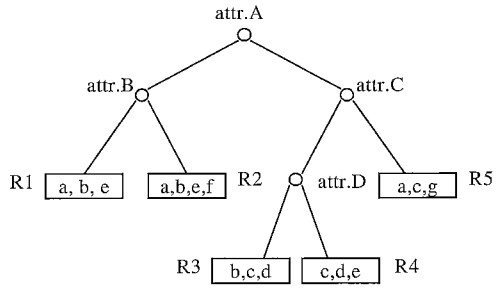
그림 3 동적 필터에 의한 해쉬 필터 생성 적용 과정

성된 해쉬 필터를 나타내며, *New S*는 릴레이션 S를 해쉬 필터에 적용하여 결합 가능성이 있는 튜플들로만 구성된 새로운 릴레이션 S를 나타낸다. 이때 $HF(B)$는 해쉬 함수에 의해 생성될 수 있는 모든 경우의 수와 그에 따른 릴레이션 R의 튜플들의 유무를 나타내는 한 비트 크기의 테이블로 형성된다. 따라서 릴레이션 S의 결합 속성값 B에 동일한 해쉬 함수를 사용하여 해쉬 필터에 적용함으로써 결합 가능성이 없는 튜플들을 제거한 새로운 릴레이션 S를 생성하게 된다.

3.2.2 정적 필터(static filter)

결합 연산을 수행하기에 앞서 모든 결합 속성값에 대한 초기 릴레이션의 해쉬 필터를 생성하는 방법으로, 결합 가능성이 없는 릴레이션의 튜플들을 최대한 제거함으로 최대한의 효율을 보장할 수 있다. 또한 모든 속성값에 대해 해쉬 필터를 생성함으로 인해 해쉬 필터 생성을 위한 초기 비용이 다소 커지긴 하지만, 초기 릴레이션 크기의 현저한 감소로 많은 시간을 요하는 디스크 입출력 시간이 크게 줄어듬으로해서 전체 성능에는 영향을 미치지 않는다. 정적 필터의 과정을 살펴보면 다음과 같다. 먼저 부쉬 트리가 생성되면 전체적인 결합 연산의 속성값이 결정되게 되고, 전체적인 속성값에 의해 모든 초기 릴레이션의 해쉬 필터를 생성한다. 각 초기 릴레이션의 모든 결합 속성값에 해쉬 함수를 적용하여 결합 가능성을 확인한다. 초기 릴레이션의 해쉬 필터가 생성되면, 각 초기 릴레이션의 해쉬 필터의 교집합에 의해 초기 릴레이션을 해쉬 필터에 적용한다. 따라서 그 결과 초기 릴레이션에서 결합 가능성이 없는 릴레이션의 튜플을 최대한 제거하게 되고, 이로써 전체 결합 릴레이션의 크기를 줄임으로써 전체 실행시간을 단축할 수 있다. 부쉬 트리가 그림 4의 (a)와 같이 생성되었을 때 정적 필터 방법에 의해 해쉬 필터를 생성하고 적용하는 과정은 그림 4와 같다.

부쉬 트리와 초기 릴레이션이 각각 (a), (b)와 같을 때, 각 초기 릴레이션에 대해 모든 결합 속성값의 해쉬 필터를 생성하면 (c)와 같다. 각 초기 릴레이션으로부터 생성된 해쉬 필터의 교집합 (d)를 구하여, 이를 각 초기 릴레이션에 적용하면 새로운 초기 릴레이션 (e)가 생성된다.



(a) 부쉬 트리

R1

a0	b0	e1
a1	b0	e1
a3	b3	e6
a4	b4	e3
a6	b6	e2
a7	b7	e5
a9	b8	e7

R2

a0	b1	e2	f0
a1	b0	e1	f1
a2	b4	e2	f2
a3	b3	e5	f3
a4	b7	e5	f4
a5	b9	e6	f5
a6	b1	e8	f6
a7	b4	e6	f7
a8	b7	e9	f8
a8	b7	e9	f8

R3

b0	c1	d3
b2	c2	d1
b2	c3	d6
b3	c4	d2
b5	c5	d3
b5	c5	d2
b2	c5	d6
b9	c7	d8
b8	c8	d1

R4

c1	d3	e2
c2	d4	e2
c2	d5	e4
c5	d6	e6
c7	d7	e8

R5

a0	c0	g1
a1	c1	g2
a3	c3	g3
a2	c4	g4
a6	c7	g5
a7	c6	g6
a8	c1	g7
a6	c9	g7

(b) 초기 릴레이션

HF_RI

	a	b	c	d
0	1	1	d	d
1	1	1	d	d
2	0	1	d	d

HF_R2

	a	b	c	d
0	1	1	d	d
1	1	1	d	d
2	1	0	d	d

HF_R3

	a	b	c	d
0	d	1	1	1
1	d	0	1	1
2	d	1	1	1

HF_R4

	a	b	c	d
0	d	d	0	1
1	d	d	1	1
2	d	d	1	1

HF_R5

	a	b	c	d
0	1	d	1	d
1	1	d	1	d
2	1	d	0	d

(c) 해쉬 필터

	a	b	c	d
0	1	1	0	1
1	1	0	1	1
2	0	0	0	1

(d) 해쉬 필터의 교집합

New R1

a0	b0	e1
a1	b0	e1
a3	b3	e6
a4	b4	e3

New R2

a1	b0	e1	f1
a3	b3	e5	f3
a9	b9	e0	f9

New R3

b0	c1	d3
b3	c4	d2
b9	c7	d8

New R4

c1	d3	e2
c7	d7	e8

New R5

a1	c1	g2
a6	c7	g5

(e) 새롭게 형성된 초기 릴레이션

그림 4 정적 필터에 의한 해쉬 필터 생성 적용 과정

4. 비용 모형(Cost Model)

새롭게 제안한 프로세서 할당 기법에 대한 비용 성능 평가를 위해 분석적 비용 모델을 제시하였으며, 이는 다음과 같은 가정 하에서 이루어진다.

(1) 컴퓨터 구조는 분산 메모리와 공유 디스크를 갖는 다중 프로세서 구조이다.

(2) 각 프로세서에서는 디스크로부터 릴레이션의 일부를 읽고 쓰기 위한 디스크 I/O 처리를 반드시 수행한다.

새롭게 제안된 프로세서 할당 기법의 성능 분석을 위한 분석적 비용 모델은 다음과 같이 구성된다. 먼저 서로 다른 방법의 해쉬 필터를 사용하였을 때의 특징을 비교하기 위해 동적 필터와 정적 필터의 비용 모델을 세우고, 다음 두 해쉬 필터 방법에 적용할 새로운 프로세서 할당 기법의 비용 모델을 구축한다.

성능 분석 비용 모델을 위한 표기법은 다음과 같다.

n_p	프로세서 수
I_{read}	디스크로 부터 한 페이지를 읽기 위한 명령어 수
I_{write}	디스크에 한 페이지를 쓰기 위한 명령어 수
I_{tuple}	메모리로 부터 한 튜플을 추출하기 위한 명령어 수
I_{build}	테이블 구축을 위한 튜플 당 명령어 수
I_{probe}	튜플 조사를 위한 튜플 당 명령어 수
I_{hash}	해쉬값을 얻기 위한 튜플 당 명령어 수
I_{apply}	한 튜플을 해쉬 필터에 적용하기 위한 명령어 수
$I_{intersection}$	해쉬 필터의 교집합을 구하기 위한 명령어 수
P_{size}	페이지 당 튜플 수
t_{pio}	페이지 당 디스크 서비스 시간(ms)
$keyA$	결합 속성 값의 크기(bytes)
TS_{Ri}	릴레이션 R_i의 한 튜플의 크기(bytes)

4.1 해쉬 필터 비용 모델

결합 연산의 수행에 있어 결합률을 높이고 비결합 가능성의 튜플을 제거하기 위한 방법으로 사용되는 해쉬 필터 사용 비용을 계산하면 다음과 같다.

4.1.1 동적 필터

모든 결합 단계에서 해쉬 필터를 사용하는 방법으로 다음의 실행비용은 할당 트리의 각 노드에서의 비용을 나타낸다. 이때 할당 트리 각 노드는 우향 트리로 구성되어 있으므로 할당 트리 각 노드의 세부 실행비용은 우향 트리의 각 노드에서 발생하는 비용을 모두 합해야

한다. 여기서 n은 우향 트리의 깊이를, N은 릴레이션의 크기를 나타낸다.

- 디스크로부터의 해당 결합 속성 값의 입력 비용

결합 연산을 수행하기 위한 릴레이션의 튜플들 중 해쉬 필터를 생성하기 위해 해당 속성 값을 디스크로부터 읽어 들인다. 이 비용은 I/O 비용이다.

$$A\_input\_I/O = \sum_{i=0}^{n} \frac{N_{Ri}}{P_{size}} \times \frac{keyA}{TS_{Ri}} \times t_{pio}$$

- 튜플의 결합 속성 추출 비용

디스크로부터 릴레이션의 해당 속성값을 페이지 단위로 읽어 들인 후 메모리의 한 페이지로부터 각 튜플의 해당 속성값을 추출한다. 이 비용은 릴레이션의 해당 속성값을 읽기 위한 CPU 비용으로 디스크로부터 한 페이지의 해당 속성값을 읽는데 필요한 CPU 비용에 메모리의 한 페이지로부터 각 튜플의 해당 속성값을 추출하는 비용의 합이다.

$$A\_input\_CPU =$$

$$\left(\sum_{i=0}^{n} \frac{N_{Ri}}{P_{size}} \times \frac{keyA}{TS_{Ri}} \times I_{read} + \sum_{i=0}^{n} N_{Ri} \times \frac{keyA}{TS_{Ri}} \times I_{attr} \right) / P_{speed}$$

- 해쉬 필터 생성/적용 비용

할당 트리의 각 노드에서는 내부 릴레이션의 각 튜플에서 추출된 속성값의 해쉬값을 생성하여 해쉬 필터를 만들고, 외부 릴레이션의 각 튜플 해당 속성값을 해쉬 필터에 적용해야 한다. 여기서 S_I은 R_O를, S_i는 우향 트리의 각 노드에서 결합된 중간 외부 릴레이션을 나타낸다.

$$A\_HF\_CPU =$$

$$\left(\sum_{i=1}^{n} N_{Ri} \times \frac{keyA}{TS_{Ri}} \times I_{hash} + \sum_{i=1}^{n} N_{Si} \times \frac{keyA}{TS_{Ri}} \times I_{apply} \right) / P_{speed}$$

- 해쉬 필터 총 사용 비용

동적 필터 방법에 의해 할당 트리의 각 노드에서 해쉬 필터를 사용하는 비용은 다음과 같다.

$$CG_{cost} = A\_input\_I/O + A\_input\_CPU + A\_HF\_CPU$$

4.1.2 정적 필터

결합 연산을 수행하기에 앞서 기본 릴레이션에 대해여 모든 결합 속성값에 해쉬 필터를 적용하는 방법으로 다음의 실행비용은 할당 트리의 각 노드에서의 실행비용을 나타낸다.

- 디스크로부터의 해당 결합 속성값의 입력 비용

결합 연산을 수행하기 위한 기본 릴레이션의 튜플에서 모든 결합 속성값에 대해 해쉬 필터를 생성하기 위해 결합 속성값을 디스크로부터 읽어 들인다. 이 비용은 I/O 비용이다. 여기서 $base$는 기본 릴레이션의 수를, $join_{attr}$는 모든 결합 속성값의 집합을 나타낸다.

$$A\_input\_I/O = \frac{\sum_{i=1}^{base} \sum_{j \in join_{attr}} \frac{N_i}{P_{size}} \times \frac{keyA_j}{TS_{Ri}} \times t_{pio}}{n_p}$$

- 튜플의 결합 속성 추출 비용

릴레이션의 모든 결합 속성값을 디스크로부터 페이지 단위로 읽어 들인 후 메모리로부터 각 튜플의 해당 속성값을 추출한다. 이 비용은 릴레이션의 모든 결합 속성값을 읽기 위한 CPU 비용으로, 디스크로부터 한 페이지의 해당 속성값을 읽는데 필요한 CPU 비용과 메모리의 한 페이지로부터 각 튜플의 해당 속성값을 추출하는 비용의 합이다.

$$A\_input\_CPU =$$

$$\frac{\left(\sum_{i=1}^{base} \sum_{j \in join_{attr}} \frac{N_i}{P_{size}} \times \frac{keyA_j}{TS_{Ri}} \times I_{read} + \sum_{i=1}^{base} \sum_{j \in join_{attr}} N_i \times \frac{keyA_j}{TS_{Ri}} \times I_{attr} \right)}{(n_p \times P_{speed})}$$

- 해쉬 필터 생성/적용 비용

기본 릴레이션의 각 튜플에서 추출된 모든 결합 속성값의 해쉬값을 생성하여 해쉬 필터를 만들고, 기본 릴레이션에서 생성된 해쉬 필터의 교집합을 구한다. 구해진 해쉬 필터의 교집합은 결합 연산 단계에서 기본 릴레이션의 튜플들을 추출하기 위해 사용된다.

$$A\_HF\_CPU =$$

$$\frac{\left(\sum_{i=1}^{base} \sum_{j \in join_{attr}} N_i \times \frac{keyA_j}{TS_{Ri}} \times I_{hash} + \#join_{attr} \times I_{intersection} \times base \right)}{(n_p \times P_{speed})}$$

- 해쉬 필터 총 사용 비용

정적 필터 방법에 의해 기본 릴레이션에서 해쉬 필터를 사용하는 비용은 다음과 같다.

$$EG_{cost} =$$

$$A\_input\_I/O + A\_input\_CPU + A\_HF\_CPU$$

4.2 결합 비용

해쉬 필터에 의해 감소된 릴레이션의 튜플들을 할당 트리에 의해 결합하는 비용으로, 디스크로부터 릴레이션을 입력하는 비용, 해쉬 테이블을 구축하고 조사하는 비용, 그리고 결과 릴레이션을 디스크에 저장하는 비용의 합이다. 이때 할당 트리의 각 노드는 우향 트리로 구성되어 있으므로 할당 트리 각 노드의 결합 비용 역시 우향 트리의 각 노드의 결합 비용을 모두 합해야 한다. 여기서 n은 우향 트리의 깊이를, N'는 해쉬 필터에 의해 감소된 릴레이션의 크기를 나타낸다.

- 해쉬 필터 적용 후 디스크로부터의 릴레이션 입력 비용

결합 연산을 수행하기 위해 디스크로 부터 해쉬 필터에 의해 감소된 릴레이션의 튜플들을 읽어 들인다. 이 비용은 I/O 비용이다.

$$R\_input\_I/O = \sum_{i=0}^{n} \frac{N'_{Ri}}{P_{size}} \times t_{pio}$$

- 릴레이션의 튜플 추출 비용

디스크로부터 릴레이션의 튜플들을 페이지 단위로 읽어 들인 후 메모리의 페이지로부터 튜플을 추출한다. 따라서 릴레이션의 튜플 추출 비용은 디스크로부터 페이지 단위로 튜플을 읽어 들이는 CPU 비용과 메모리에서 튜플들을 추출하는 비용의 합이다.

$$R\_input\_CPU =$$

$$(\sum_{i=0}^{n} \frac{N'_{Ri}}{P_{size}} \times I_{read} + \sum_{i=0}^{n} N'_{Ri} \times I_{tuple}) / P_{speed}$$

- 해쉬 테이블 구축 및 조사 비용

추출 된 튜플에 의해 해쉬 테이블을 구축하고, 외부 릴레이션의 조사 과정을 수행한다. 여기서 S_1은 R_0를, S_i는 우향 트리의 각 노드에서 결합된 중간 외부 릴레이션을 나타낸다.

$$R\_join\_CPU =$$

$$(\sum_{i=1}^{n} N'_{Ri} \times I_{build} + \sum_{i=1}^{n} N'_{Si} \times I_{probe}) / P_{speed}$$

- 결과 릴레이션 생성 비용

할당 트리의 각 노드에서 생성된 결과 릴레이션을 디스크에 저장한다. 이때 필요한 CPU 비용은 다음과 같다.

$$R\_output\_CPU = (\frac{N_{Sn+1}}{P_{size}} \times I_{write}) / P_{speed}$$

- 결과 릴레이션의 디스크 저장 비용

할당 트리의 각 노드에서 생성된 결과 릴레이션을 디스크에 저장하기 위한 I/O 비용은 다음과 같다.

$$R\_output\_I/O = \frac{N_{Sn+1}}{P_{size}} \times t_{pio}$$

- 결합 총 비용

할당 트리의 각 노드에서 결합 연산을 수행하기 위한 비용은 다음과 같다.

$$Join_{cost} = R\_input\_I/O + R\_input\_CPU + R\_join\_CPU + R\_output\_CPU + R\_output\_I/O$$

4.3 총 비용

해쉬 필터를 사용한 후 할당 트리에 의해 결합 연산을 수행한다.

이때 동적 필터의 해쉬 필터 방법을 사용하는 경우는 할당 트리의 각 노드에서 각각 해쉬 필터 사용과 결합 연산을 모두 수행한다. 따라서 해쉬 필터 방식으로 동적 필터를 이용할 경우 할당 트리 전체의 총 비용은 다음과 같다. 여기서 m은 할당 트리의 깊이를, $equal_{level}$은 할당 트리에서 동일한 깊이를 갖는 노드들의 집합을 의미한다. 또한 $\#P(S_x)$는 할당 트리의 노드 S_x에 할당된 프로세서의 수를 나타낸다.

$$Total_{cost} =$$

$$\sum_{level=1}^{m} \max_{x \in equal_{level}} (\frac{CG_{cost}(S_x) + Join_{cost}(S_x)}{\#P(S_x)})$$

반면, 정적 필터를 사용하는 경우는 초기에 해쉬 필터를 모든 결합 속성값에 대해 적용하여 기본 릴레이션의 크기를 줄인 후 할당 트리에 의해 결합 연산을 수행한다. 따라서 정적 필터의 해쉬 필터 방식을 사용할 경우 할당 트리 전체의 총 비용은 해쉬 필터 이용 비용과 결합 연산 비용의 합으로 나타낼 수 있다. 이 경우 할당 트리 전체에 대한 총 비용은 다음과 같다.

$$Total_{cost} =$$

$$EG_{cost} + \sum_{level=1}^{m} \max_{x \in equal_{level}} (\frac{Join_{cost}(S_x)}{\#P(S_x)})$$

5. 성능 분석과 비교

제안한 동적 프로세서 할당 기법을 이용한 해쉬 결합 알고리즘을 기존의 정적 프로세서 할당 기법을 이용한 해쉬 결합 알고리즘과 비교하여 성능 향상 정도를 보이고, 해쉬 필터에 따른 동적 프로세서 할당 기법과 정적 프로세서 할당 기법의 성능 변화를 알아본다.

위에서의 비용 모형에 기반을 두고 분석하였으며, 분석 모델에 사용된 매개 변수는 다음과 같다.

n_p	32	개
I_{read}	5,000	inst.
I_{write}	5,000	inst.
I_{tuple}	300	inst.
I_{build}	100	inst.
I_{probe}	200	inst.
I_{hash}	100	inst.
I_{apply}	200	inst.
P_{size}	40	tuples
t_{pio}	20	ms
$keyA$	13	bytes

그림 5는 새롭게 제안한 동적 프로세서 할당 기법을 이용한 해쉬 결합 알고리즘(Ours)의 전체 실행시간을 기존의 정적 프로세서 할당 기법을 이용한 알고리즘(Hsiao)과 비교하였다. 동적 프로세서 할당 기법을 이용하는 경우 할당 트리 각 노드의 실행결과를 정확히 파악하여 다음 결합 연산의 기본 연산으로 사용함으로써 정확한 실행시간 계산이 가능하다. 그러나 정적 프로세서 할당 기법을 이용하는 경우 할당 트리 각 노드에서

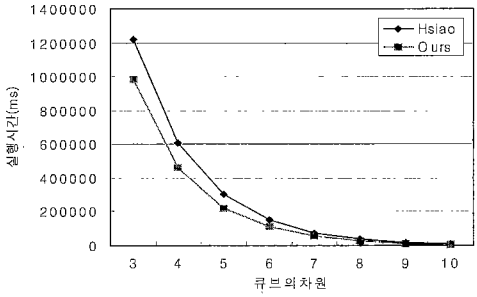


그림 5 동적 프로세서 할당 기법과 정적 프로세서 할당 기법의 비교

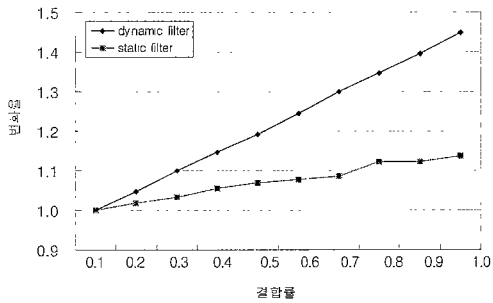


그림 7 각 해쉬 필터에 대한 결합률 변화에 따른 성능 비교

는 초기 릴레이션의 정보만으로 실행시간을 예측하며, 비용 모델 역시 할당 트리 각 노드의 중간 결과를 초기 릴레이션의 크기에 의해 임의의 축소비율을 적용하여 계산한다. 따라서 실제 두 알고리즘의 실행시간은 그림 5에서 보이는 결과보다 좀 더 큰 차이를 보이게 되고, 동적 프로세서 할당 기법을 이용한 알고리즘이 보다 더 효율적이라는 것을 입증할 수 있다.

그림 6은 결합 가능성이 없는 튜플을 미리 제거하여 결합률을 높이고 전체 실행시간을 줄이기 위한 방법으로 해쉬 필터를 이용한 결과를 보이고 있다. 동적 필터와 정적 필터 모두 동적 프로세서 할당 기법을 이용한 해쉬 결합 알고리즘이 정적 프로세서 할당 기법을 이용한 알고리즘보다 더 나은 성능을 보이는 것을 볼 수 있다. 그러나 동적 필터와 정적 필터를 비교해 보면 정적 필터에 비해 동적 필터가 더 많은 실행시간을 필요로 하는 것을 알 수 있다. 이는 동적 필터의 경우, 결합 연산을 수행하기에 앞서 모든 결합 속성값에 대해 해쉬 필터를 적용함으로써 결합 가능성이 없는 튜플들을 모두 제거한 정적 필터에 비해 모든 릴레이션의 크기가 큼으로, 연산시간에 비해 상대적으로 많은 처리시간을

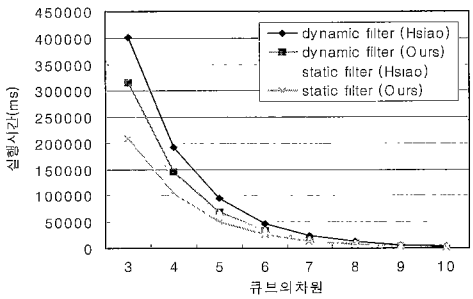


그림 6 해쉬 필터에 따른 동적 프로세서 할당 기법과 정적 프로세서 할당 기법 비교

요구하는 디스크 입출력 시간이 릴레이션의 크기에 비례하여 포함되기 때문이다. 즉, 모든 결합 속성값에 해당하는 해쉬 필터를 생성하는 시간이 동적 필터에 의한 해쉬 필터의 사용으로 발생한 릴레이션의 크기 변화에 따른 실행시간보다 적음을 나타낸다.

그림 7은 각 해쉬 필터에 대한 결합률 변화에 따른 실행시간 결과를 보이고 있다. 동적 필터의 경우는 결합률이 낮아 중간 결과 릴레이션의 크기가 작을 때 실행시간이 적게 걸리는 것을 볼 수 있다. 즉 할당 트리 각 노드에서 이루어지는 해당 결합 속성값에 대해서만 해쉬 필터를 생성하므로 전체 릴레이션의 크기에 비해 결합률이 적은 경우 릴레이션의 크기가 급격히 줄어듬으로 인해 보다 효율적임을 확인할 수 있다.

그러나 결합률이 높은 경우는 중간 결과 릴레이션 각각에 대해 해쉬 필터를 새롭게 생성해야 하므로 릴레이션의 크기만큼 실행시간이 많이 걸리게 된다. 따라서 중간 결과 릴레이션의 크기, 즉 결합률에 따라 실행시간이 급격히 변화하는 것을 알 수 있다. 이에 반해 정적 필터의 경우는 결합 연산을 수행하기에 앞서 모든 결합 속성값에 대해 해쉬 필터를 적용함으로써 결합 가능성이 있는 튜플들 만으로 결합 연산을 수행한다. 따라서 중간 결과 릴레이션의 결합률이 동적 필터에 비해 높게 되므로 결합률의 변화에 크게 영향을 받지 않게 된다.

6. 결 론

본 논문에서는 부쉬 트리를 할당 트리로 변환한 후 결합 연산을 수행하면서 실제 실행시간을 동적으로 계산하고 그 결과에 의해 실시간에 프로세서를 할당하는 동적 프로세서 할당 기법을 이용한 파이프라인 해쉬 결합 알고리즘을 제안하였다. 기존의 정적 프로세서 할당 기법에 의한 파이프라인 해쉬 결합 알고리즘에서도 부

쉬 트리를 할당 트리로 변환하여 수행하였으나, 프로세서를 할당하는 과정에서 초기 릴레이션의 기본 정보만을 이용하여 누적 실행비용을 계산하고, 그 값에 의해 미리 프로세서를 할당함으로써 정확한 실행시간을 예측할 수 없었다. 즉 할당 트리 한 노드의 결합 연산 결과는 다른 노드의 입력 릴레이션으로 사용되기 때문에 그 노드의 실행결과의 크기에 따라 다음 결합 연산의 실행시간을 크게 좌우하므로 결합 연산을 수행하기에 앞서 미리 모든 결합 연산에 필요한 프로세서를 할당하는 것은 비효율적이다. 따라서 본 논문에서 제안한 동적 프로세서 할당 기법에서는 할당 트리 각 노드의 실행결과를 포함한 결합 과정 중의 정보를 다음 노드의 실행시간 예측에 충분히 반영하여 할당 트리의 동일한 깊이의 노드들에게 프로세서를 비례적으로 분배하였다. 이로써 각 프로세서의 유휴시간을 최소화하고 주어진 프로세서를 최대한 활용함으로써 전체적인 실행시간을 충분히 줄이고 병렬 처리의 효과를 최대화하였다.

또한 전체적인 질의 실행시간을 줄이기 위하여 결합 가능성이 없는 튜플들을 제거한 후 결합 연산을 수행할 수 있도록 해쉬 필터 기법을 이용하였다. 해쉬 필터 기법으로는 결합 연산을 수행하기에 앞서 모든 결합 속성 값에 대해 해쉬 필터를 생성하는 정적 필터와 각각의 결합 연산 직전에 해쉬 필터를 생성하는 동적 필터를 사용하였다. 그 결과, 정적 필터의 사용이 동적 필터를 사용할 경우에 비해 결합 가능성이 있는 튜플들 만의 추출로 최대한의 결합률을 보이게 되고 이로써 초기 릴레이션의 크기를 충분히 줄임으로써 결합 연산의 실행비용을 크게 줄일 수 있었다.

참 고 문 헌

[1] Hui-I Hsiao, Ming-Syan Chen, Philip S. Yu, "Parallel Execution of Hash Joins in Parallel Databases," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 8, pp. 872 -883, Aug. 1997.

[2] D. Schneider, D.J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," *Proc. 16th Int'l Conf. Very Large Data Bases*, pp. 469-480, Aug. 1990.

[3] Ming-Syan Chen, Ming-Ling Lo, Philip S. Yu, Honesty C. Young, "Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins," *Proc. 18th Int'l Conf. Very Large Data Bases*, pp. 15-26, Aug. 1992.

[4] Ming-Syan Chen, Hui-I Hsiao, Philip S. Yu, "Applying Hash Filters to Improving the Execution of Bushy Trees," *Proc. 14th Int'l Conf. Very Large Data Bases*, 1993.

[5] D.J. DeWitt, J. Gray, "Parallel Database Systems : The Future of High Performance Database Systems," *Comm. ACM*, vol. 35, no. 6, pp. 85-98, June 1992.

[6] Ming-Syan Chen, Philip S. Yu, K. L. Wu, "Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries," Proc. Eighth Int'l Conf. Data Eng., pp. 58-67, Feb. 1992.

[7] Mingling Lo, Ming-Syan Chen, C. V. Ravishankar, and Philip S. Yu, "On Optimal Processor Allocation to Support Pipelined Hash Joins," *Proc. ACM SIGMOD*, pp.69-78, May 1993.

[8] Ming-Syan Chen, Mingling Lo, Philip S. Yu, and Honesty C. Young, "Applying Segmented Right-Deep Trees to Pipelining Hash Joins," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995.

이 규 옥

1985년 충남대학교 계산통계학과 졸업. 1987년 충남대학교 계산통계학과 석사. 1987년 ~ 1989년 한국동력자원연구소 위촉연구원. 1989년 ~ 현재 한국기계연구원 선임연구원. 1995년 3월 ~ 현재 아주대학교 컴퓨터공학과 박사과정. 관심분야는 병렬데이타베이스, 컴퓨터그래픽스, 시스템 통합임.

이 동 련

1999년 아주대학교 정보 및 컴퓨터공학부 학사. 현재 아주대학교 정보통신전문대학원 석사과정. 관심분야는 병렬처리, 보안관리, 침입탐지

원 영 선

1993년 경기대학교 전자계산학과 이학사. 1995년 경기대학교 전자계산학과 이학석사. 1996년 현재 아주대학교 컴퓨터공학과 박사과정. 관심분야는 병렬처리, 병렬 데이타베이스, 병렬 알고리즘.

홍 만 표

서울대학교 자연과학대학 계산통계학과(1981) 석사 서울대학교 자연과학대학 계산통계학과(1983) 박사 서울대학교 자연과학대학 계산통계학과(1991) 1983년 ~ 1985년 울산공과대학 전자계산학과 전임강사. 1985년 ~ 현재 아주대학교 정보 및 컴퓨터공학부 교수. 1993년 ~ 1994년 미네소타대학 전자공학과 교환교수. 관심분야는 병렬처리