다중 해시 조인의 파이프라인 처리에서 분할 조율을 통한 부하 균형 유지 방법

(A Load Balancing Method using Partition Tuning for Pipelined Multi-way Hash Join)

문 진 규 [†] 진 성 일 [††] 조 성 현 [†††]

(Jin Gue Moon) (Seong Il Jin) (Sung Hyun Cho)

요 약 Shared nothing 다중 프로세서 환경에서 조인 어트리뷰트의 자료 불균형(data skew)이 파이프라인 해시 조인 연산의 성능에 주는 영향을 연구하고, 자료 불균형을 대비하여 적재부하를 Round-robin 방식으로 정적 분할하는 방법과 자료분포도를 이용하여 동적 분할하는 두 가지 파이프라인 해시 조인 알고리즘을 제안한다. 해시 기반 조인을 사용하면 여러 개의 조인을 파이프라인 방식으로 처리할 수 있다. 다중 조인의 파이프라인 방식 처리는 조인 중간 결과를 디스크를 통하지 않고 다른 프로세서에게 직접 전달하므로 효율적이다. Shared nothing 다중 프로세서 구조는 대용량 데이타베이스를 처리하는데 확장성은 좋으나 자료 불균형 분포에 매우 민감하다. 파이프라인 해시 조인 알고리즘이 동적 부하 균형 유지 메커니즘을 갖고 있지 않다면 자료 불균형은 성능에 매우 심각한 영향을 줄 수 있다. 본 논문은 자료 불균형의 영향과 제안된 두 가지 기법을 비교하기 위하여 파이프라인 세그먼트의 실행 모형, 비용 모형, 그리고 시뮬레이터를 개발한다. 다양한 파라미터로 모의 실험을 한 결과에 의하면 자료 불균형은 조인 선택도와 릴레이션 크기에 비례하여 시스템 성능을 떨어뜨림을 보여준다. 그러나 제안된 파이프라인 해시 조인 알고리즘은 다수의 버켓 사용과 분할의 조율을 통해 자료 불균형도가 심한 경우에도 좋은 성능을 갖게 한다.

키워드 : 병렬 질의 처리, 파이프라인 해시 조인, 자료 불균형, 부하 균형 유지

Abstract We investigate the effect of the data skew of join attributes on the performance of a pipelined multi-way hash join method, and propose two new hash join methods in the shared-nothing multiprocessor environment. The first proposed method allocates buckets statically by round-robin fashion, and the second one allocates buckets dynamically via a frequency distribution. Using hash-based joins, multiple joins can be pipelined so that the early results from a join, before the whole join is completed, are sent to the next join processing without staying in disks. Shared nothing multiprocessor architecture is known to be more scalable to support very large databases. However, this hardware structure is very sensitive to the data skew. Unless the pipelining execution of multiple hash joins includes some dynamic load balancing mechanism, the skew effect can severely deteriorate the system performance. In this paper, we derive an execution model of the pipeline segment and a cost model, and develop a simulator for the study. As shown by our simulation with a wide range of parameters, join selectivities and sizes of relations deteriorate the system performance as the degree of data skew is larger. But the proposed method using a large number of buckets and a tuning technique can offer substantial robustness against a wide range of skew conditions.

Key words : Parallel Query Evaluation, Pipelined Hash Join, Data Skew, Load Balance

† 정 회 원 : 국방과학연구소 선임연구원
jingue@add.re.kr
†† 정 회 원 : 충남대학교 정보통신공학부 컴퓨터과학과 교수
sijin@cs.chungnam.ac.kr
††† 종신회원 : 홍익대학교 전자전기컴퓨터공학부 교수
scho@wow.hongik.ac.kr
논문접수 : 2001년 7월 9일
심사완료 : 2002년 2월 20일

1. 서 론

관계형 데이타베이스 시스템에서 조인은 실행 비용이 가장 큰 연산자인데 릴레이션의 크기와 질의의 복잡도가 커짐에 따라 실행 비용은 더욱 증가한다. 다중(multi-way)조인 질의는 2-way 조인들을 연결시킨 조

인질의로서 실행 비용이 크고 질의 실행 계획 작성이 복잡하다[1,2,5]. 미래의 데이타베이스 성능 향상을 위해서 병렬 처리는 제 일의 해결책으로 인식되어져 왔다. 병렬 데이타베이스 머신의 성공에 이어, 최근에는 상용 저가의 CPU, 메모리, I/O장치 등이 고속 망에 연결된 상용 병렬 컴퓨터가 등장하면서 다중 프로세서 기반 병렬 데이타베이스의 발전 가능성을 더욱 밝게 하고 있다[3]. 2-way 조인 연산에 대한 연산자 내 병렬성(parallelism)을 활용하는 연구[4]와 함께 최근에는 m-way 다중 조인 연산에 대한 연산자 간 병렬성을 활용하는 연구가 활발하다[6,7,9,10]. 2-way 조인에서 해시 기반 조인 방법들은 여러 가지 조인 방법들 중 가장 성능이 좋은 방법으로 평가를 받았으며[4], 특히 파이프라인 처리가 가능하다. [8,9,10,11]에서는 다중 해시 조인의 파이프라인 처리에서 프로세서의 할당 방법, 질의 실행 계획에 부시 트리(bushy tree)의 활용 등에 대하여 연구하였다.

병렬 처리의 기본 개념은 자료를 모든 저장 장치들로부터 병렬로 접근하고 자료가 모든 처리 노드들에 의하여 병렬로 처리될 수 있도록 하는 것이다. 병렬성을 활용하려면 하나의 질의 또는 여러 개의 질의들이 여러 개의 작업들로 나누어지고, 각 작업들이 모든 프로세서들 상에서 병렬로 실행될 수 있어야 한다. 병렬 실행의 효과성을 높이기 위해서는 작업들을 프로세서들에게 균등히 배분시키고 동시에 협력 작업이나 동기화 오버헤드는 최소화되어야 한다[1,3].

조인 질의들의 병렬 처리 성능개선의 방해요소 중 하나는 조인할 어트리뷰트(attribute)에 내재하는 자료 값의 불균형 분포(data skew)이다[13,14,15]. 실제 데이타베이스에서 주어진 어트리뷰트가 갖는 값들 중 어떤 값들이 다른 값들에 비해 더 많이 나타나는 것을 쉽게 발견할 수 있으며 이러한 불균형을 자료의 불균형 분포라고 부른다[13]. 이것은 자료에 내재되어 있는 문제이며 자료의 접근 유형과는 무관하다. 병렬 컴퓨터 구조가 많은 프로세서들을 탑재하였을 때 자료의 불균형 분포는 어떤 프로세서들은 처리할 많은 투플(tuple)들로 인해 매우 바쁜 반면 처리할 투플이 적은 프로세서는 많은 시간 유휴상태에 머물게 된다. 작업 부하 불균형을 일으키는 다른 요인으로 처리 시간의 통계적 성격, 조인의 선택도(selectivity) 등 실행 시간에 영향을 주는 파라미터는 많이 있다[1,2,13].

해시 기반 조인 방법에서 자료의 불균형 분포를 대비하여 작업 부하의 균형을 유지하기 위한 많은 연구가 있었다[13,14,15]. 그러나 주로 연산자 내(intra-operation) 병렬성을 활용하는 환경에서의 부하 균형 유지에 대한 연구이며 다중 해시 기반 조인의 파이프라인 처리에 있어서의 연구는 미흡하다. 파이프라인 방식의 처리 환경에서는 기존의 2-way 조인의 작업 균형 유지 기법들을 고려하여야할 뿐만 아니라 연산자 간(inter-operation)에 발생하는 중간결과의 불균형 분포도 고려하여야 한다. 본 논문은 Shared-nothing 컴퓨터 구조의 Ad-hoc m-way 해시 조인의 파이프라인 처리에서 부하 균형의 동적 유지 방법에 대하여 연구한다. 모든 처리 노드는 독립적으로 프로세서, 메모리, 그리고 디스크를 갖고 있고, 고속 망에 상호 연결되어 있다. 다중 조인에서 조인 어트리뷰트에 대한 인덱스는 없다고 가정한다. 먼저 조인 어트리뷰트 값의 불균형 분포가 다중 해시 조인의 파이프라인 처리에 주는 영향을 분석하고, 이 영향을 최소화하는 부하의 균형 유지 메커니즘을 갖춘 새로운 파이프라인 해시 조인 알고리즘을 제안한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 관련 연구로서 다중 조인 연산의 최적화 및 병렬화에 대한 연구 활동을 소개하고 기존 연구의 문제점을 살펴본다. 3장에서는 파이프라인 해시 조인 환경에서 조인 어트리뷰트의 자료 불균형 분포를 정의하고, 불균형 분포를 대비한 새로운 파이프라인 해시 조인 알고리즘에 대하여 설명한다. 4장에서는 모의 실험을 위해 개발한 실행모형, 비용모형 그리고 부하모형에 대하여 설명한다. 5장에서는 실험 결과에 대해 설명하고, 6장에서 결론 및 연구 방향에 대해 설명한다.

2. 관련 연구

다중 조인 연산의 최적화 및 병렬화에 대한 여러 연구가 진행되었다[1,2,10,11]. 파이프라인 해시 조인의 성능 개선 연구는 2-way 해시 조인의 병렬화 연구와 다중 해시 조인의 최적화 및 병렬화 연구로 분류할 수 있다. 관련 연구에 대하여 간략히 설명한 후, 기존 파이프라인 해시 조인[9,10]의 문제점을 언급한다.

2-way 해시 조인을 단일 프로세서 환경과 다중 프로세서 환경으로 구분하여 개략적으로 설명하면 다음과 같다[4]. 단일 프로세서 환경에서 조인 연산 $R \bowtie S$(여기서, $|R| \leq |S|$)이 있을 때, R의 각 투플의 조인 어트리뷰트를 해시(hash)하여 메모리 내에 해시테이블을 만든다(R을 내부 릴레이션 또는 테이블 구축용 릴레이션이라고 함). 그리고 난 후 S(외부 릴레이션 또는 투플 검사용 릴레이션)의 각 투플의 조인 어트리뷰트를 해시하여 메모리 내에 있는 R의 해시테이블을 검사한다. 대응이 되면 두 투플을 결합하여 결과 릴레이션에 추가한다.

다중 프로세서 환경에서는 R과 S가 어떤 키 값에 의해 N개의 처리노드에 수평으로 분산 저장되어 있다. R_i, S_i 각각을 어떤 처리노드$_i$에 저장된 R과 S의 부 릴레이션(sub-relation)이라 하자($2 \le i \le N$). 다중 프로세서 환경에서의 2-way 병렬 해시 조인은 조인 연산을 프로세서 수만큼 분할 처리할 수 있도록 두 개의 단계로 구성된다: 분리단계(split phase), 조인단계(join phase). 분리단계에서 각 처리 노드는 R_i와 S_i를 처리 노드의 수만큼 분할하여 같은 범위의 해시 값을 갖는 R과 S의 분할(partition)이 동일한 처리 노드에 할당되도록 분배시킨다. 조인단계에서 각 처리 노드는 독립적으로 단일 프로세서 환경과 같은 방식으로 조인한다. 대표적인 조인 방법으로 Simple 해시조인, Grace 해시조인, Hybrid 해시조인이 있다[4,15].

다중 프로세서 환경의 2-way 병렬 해시 조인에서 분리단계는 피 연산자 릴레이션들을 조인 어트리뷰트 해시 값의 범위별로 여러 개의 분할로 나누어 조인단계에서 각각의 분할을 프로세서별로 해시 조인한다. 만약 조인 어트리뷰트의 값이 균등분포(uniform distribution)로 되어 있다면 분할들의 크기는 비슷할 것이다. 그러나 조인 어트리뷰트의 특정 값들의 빈도가 다른 값들에 비하여 많은 경우는 가장 큰 분할을 조인하는 프로세서가 존재하게 되므로 작업 부하가 달라진다. [15]에서는 2-way 해시 조인의 병렬 처리에서 분할(partition)의 조율(tuning)을 통한 작업 부하 균형 유지에 대해 연구하였다. [15]의 ABJ(Adaptive Load Balancing Parallel Hash Join) 방법은 기존의 Grace 해시 조인 알고리즘에 부하 균형 유지 단계를 삽입한 것이다: 분리단계, 조율단계(tuning phase), 조인단계. 분리단계에서 각 프로세서$_i$는 부 릴레이션 R_i와 S_i 각각을 많은 수의 부 버켓(sub-bucket)들로 분할시킨다. 분리 단계에서 $|R_i|$ 또는 $|S_i|$가 메모리 용량을 넘는 경우는 부 버켓들이 지역 디스크로 다시 쓰여진다. 조율단계에서는 분리단계에서 수집된 버켓들의 크기 정보를 사용하여 분할이 균등히 배분되도록 하기 위해 각 처리노드는 자신이 만든 R_i와 S_i의 부 버켓들의 크기 정보를 미리 지정된 중재자 처리 노드에게 전달한다. 중재자 처리 노드는 피 연산자 릴레이션 R과 S에 대하여 전역적인 버켓 크기 정보를 수집하고 먼저 "Largest sub-bucket retaining strategy"에 의하여 버켓들을 처리 노드에게 할당한다(bucket mapping). 이 분할의 할당 방법은 부 버켓들 중 가장 큰 크기의 부 버켓을 갖고 있는 처리노드에게 분할을 할당하여 통신과 디스크 입출력 부담을 줄이려는 방법이다. 이 방법으로 버켓들을 처리 노드들에게 할당시킨 후 아직 할당이 안된 버켓들은 "Best-fit decreasing strategy"로 처리 노드들에게 할당시킨다. 이 방법은 현재 할당된 분할의 크기와 새로 할당될 버켓의 크기를 합한 것이 최소가 되는 처리 노드에게 버켓을 할당하는 방법이다. 조율단계에 의하여 분할벡터가 만들어진다[14,15]. 예를 들어 처리 노드가 3개 있고 버켓의 수를 p로 하였을 때 분할벡터 $\langle b_1, b_2, b_3 \ldots, b_p \rangle = \langle 0,1,2,\ldots,1 \rangle$ 인 경우, 버켓 b_1은 처리노드#0에게, 버켓 b_2는 처리노드#1에게, 버켓 b_3는 처리노드#2에게, ..., 버켓 b_p는 처리노드#1에게 할당된다. 중재자 처리 노드는 이 분할벡터를 모든 처리노드들에게 방송하여 버켓의 재분배가 시작된다. 버켓의 재분배가 종료하면 각 처리 노드는 자신의 분할에 대해 독립적으로 해시 조인한다. [15]에서는 기존 해시 조인 방법에 분할 조율 방법을 도입한 새로운 ABJ 알고리즘을 제안하여 자료의 불균형에 대해 효과적인 부하 균형을 유지하였다.

다중 해시 조인의 최적화 및 병렬화 관련 연구에 대하여 설명한다. 해시 기반 조인을 사용하면 여러 개의 조인을 파이프라인 방식으로 병렬 처리할 수 있게 되어, 전체 조인이 완료되기 전에 이전 조인의 중간 결과를 다음 조인의 입력으로 보낼 수 있으므로 질의 실행 시간을 줄일 수 있다[5,8,9,10]. 질의 최적화기는 주어진 질의에 대해 여러 가지 질의 실행 계획을 만들고, 그 중 최소 비용으로 판단되는 계획을 택하여 질의 실행 트리를 만든다. 질의 실행 트리는 조인 실행 순서가 명시된 이진 트리로서 좌향트리(left-deep tree), 우향트리(right-deep tree), 부시트리(bushy tree)가 있다[5,10]. 이 중에서 우향트리와 부시트리는 파이프라인 방식의 병렬 처리를 가능하게 한다. 우향트리는 구조가 단순하여 비교적 쉽게 질의 계획을 만들 수 있으나 부시트리보다 질의 계획 생성 때 융통성이 적다. [9]에서는 질의 계획 생성 때 세그먼트(segment)로 불리는 파이프라인 처리의 기본 단위를 구분하여 단순 우향트리를 사용하는 방법보다 개선된 질의 최적화 방법에 대하여 연구하였다. [10]에서는 질의 계획 생성 때 우향트리보다 좀 더 융통성 있는 부시트리를 사용하여 더욱 최적화된 질의 실행 트리를 만들었고, 할당트리를 사용하여 실행 동기화가 되도록 프로세서를 할당하였다. [11]에서는 검사용 릴레이션의 페이지 읽기 과정에 수반된 지연을 줄이고 조인 선택도와의 관계를 연구하여 개선된 프로세서 할당 방법을 연구하였다. [9,10,11]의 파이프라인 처리 단위는 파이프라인 세그먼트이다[9].

다중 해시 조인의 최적화 및 병렬화 관련 연구[2,5,6,8,9,10,11]를 보면, 질의 실행 트리의 형태가 선형트리에

서 부시트리로 구조가 점점 복잡해지고 있다. 이 연구들은 최적화를 위한 최선의 노력으로서 새로운 휴리스틱 방법들의 개발이다. 질의 실행 트리가 복잡해지면서 병렬화 기법들은 단일 조인 내부의 부하 균형뿐 아니라 조인 연산 간의 실행 동기화 문제도 고려하여야 한다. 그러나 기존의 파이프라인 해시 조인[9]은 자료 불균형 문제를 고려하지 않고 있으므로, 단일 조인 내부의 부하 불균형과 조인 연산 간의 실행 시간 차이로 인하여 전체 질의 실행 지연, 질의 최적화기의 성능 저하 등 매우 심각한 문제를 갖고 있다.

3. 파이프라인 처리 환경에서의 부하 균형 유지 방법

3.1 기존의 파이프라인 해시 조인

컴퓨터구조는 Shared nothing 구조로 가정한다. 처리 노드마다 지역 메모리와 디스크가 있고 이러한 처리 노드들이 고속 망으로 연결되어 있다. 혼돈이 되지 않는 한 여기서부터는 처리 노드와 프로세서를 혼용하기로 한다. 조인은 Equi-join 이며 조인 어트리뷰트는 미리 인덱스 되어 있지 않다고 가정한다. 모든 테이블 구축용 릴레이션의 크기는 하나의 스테이지(stage)에 참여하는 처리 노드의 메모리 크기 합보다 작으며 버킷 오버플로우는 일어나지 않는다고 가정한다. 스테이지에 한 개의 처리 노드가 할당될 수도 있으므로 5장의 실험 환경에서는 하나의 테이블 구축용 릴레이션의 크기가 처리 노드의 메모리 크기보다 작다고 가정한다.

[10]에서는 다중 해시 조인 질의 처리를 위해 그림1과 같이 부시(bushy) 실행 트리를 하나의 노드(파이프라인 세그먼트)가 우향 이진 트리로 구성된 할당 트리를 만든 후 프로세서를 할당하는 방법을 제안하였다.

파이프라인 세그먼트(이하 세그먼트)는 여러 개의 스테이지(stage, 2-way 병렬 해시조인)가 순차적으로 연결된 파이프라인 처리 구조를 갖고 있다. 그림1과 같이 구분된 세그먼트들은 각각 파이프라인 방식에 의해 아래에서 위로 처리되며, 파이프라인 처리의 기본 단위가 된다. 스테이지 수가 k개인 한 세그먼트의 파이프라인 처리[9]는 테이블 구축 단계와 투플 검사 단계로 구성된다. 편의상 하나의 스테이지에 두 개 이상의 프로세서가 할당되어 있다고 가정하고, 세그먼트에 할당된 프로세서의 개수 $N = \sum_{i=1,..,k} n_i$ 이다. 여기서 n_i는 스테이지$_i$에 할당된 프로세서 개수. 스테이지에 한 개의 프로세서가 할당된 경우 스테이지 내에서 분할 과정은 없다.

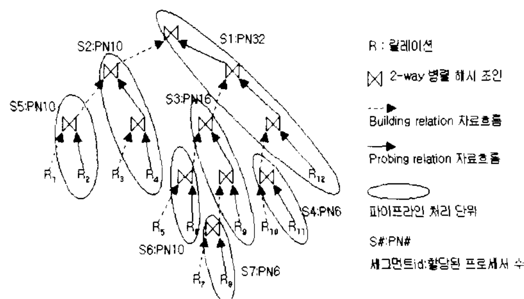


그림 1 부시 실행 트리(bushy execution tree)와 세그먼트

<u>테이블구축 단계(table building phase):</u>

N개의 프로세서가 하나의 세그먼트에 할당된 경우 모든 스테이지의 해시테이블들이 다음과 같은 과정으로 병렬로 구축된다.

step 1 : 모든 프로세서$_j$(j=1,..,N)는 디스크에서 해시 테이블 구축용 릴레이션 Ri(i=1,..,k)의 부 릴레이션 Ri_{sub}(sub=1,..,n_i)을 읽어 해시 부 테이블을 구축한다. 어떤 스테이지$_i$(i=1,..,k)에 할당된 프로세서$_j$가 해시테이블 구축용 릴레이션 Ri_{sub}의 한 블록을 읽으면 이 블록 내 투플들을 스테이지$_i$의 분할 함수로 분할한 후 각 분할을 미리 정의된 분할 벡터에 의하여 스테이지$_i$의 해당 프로세서에게 보낸다.

step 2 : 스테이지$_i$의 모든 프로세서는 자신의 분할을 입력으로 하여 해시 부 테이블(hash sub-table)을 구축한다.

<u>투플검사 단계(tuple probing phase):</u>

테이블 구축 단계가 종료된 후 다음과 같이 파이프라인 처리가 진행된다.

step 1 : 모든 프로세서$_j$(j=1,..,N)는 디스크에서 외부 릴레이션(우향 이진 트리의 맨 우측 잎노드) S의 부 릴레이션 S_{sub}(sub=1,..,N)을 블록단위로 읽는다. 각 블록은 첫 번째 스테이지의 분할함수로 분할되어 첫 번째 스테이지의 분할 벡터에 의해 지정된 해당 프로세서(첫 번째 스테이지의 참여 프로세서들 중 하나)에게 보내진다.

step 2 : 스테이지$_i$(i=1,..,k-1)의 각 프로세서$_j$는 스테이지$_i$에 들어오는 투플 검사용 릴레이션 중 자신에게 입력되는 부 릴레이션 투플들을 하나씩 해시하여 테이블 구축 단계에서 준비된 해시 부 테이블에 대응시켜 조인한다. 조인의 결과(중간결과)는 투플단위로 스테이지$_{i-1}$의 분할 함수로 분할한 후 각 분할을 스테이지$_{i-1}$의 분할 벡터가 지시하는 대로 스테이지$_{i-1}$의 해당 프로세서에게 보낸다.

step 3 : 마지막 스테이지의 각 프로세서는 스테이지

에 들어오는 투플 검사용 릴레이션 중 자신에 배정된 부 릴레이션의 투플들을 하나씩 해시하여 테이블 구축 단계에서 준비한 해시 부 테이블에 대응시켜 조인한다. 조인 결과(세그먼트의 최종 결과)는 지역 디스크에 블록 단위로 쓴다.

세그먼트의 파이프라인 처리에 대한 예를 들면 그림2와 같다.

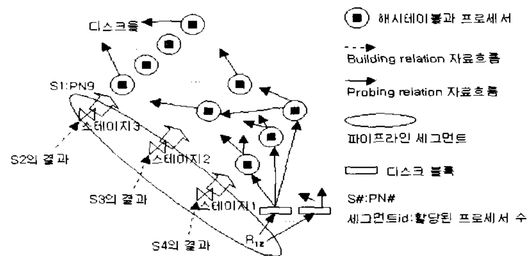


그림 2 파이프라인 세그먼트의 실행

설명을 단순화하기 위해 그림1에서 S1의 선행 세그먼트 S2, S3, 그리고 S4의 실행이 모두 완료되었고, 세그먼트 S1에 9개의 프로세서가 할당되어 있다고 가정한다. 릴레이션 R_{12}는 처리 노드의 지역 디스크에 어떤 키 값에 의해 분산 저장되어 있다고 가정한다. 본 논문에서는 릴레이션 배치 문제를 다루지 않으며, 초기 릴레이션의 분할배치는 균등분포를 가정한다.

처리 노드의 구성요소(프로세서, 메모리, 디스크, 통신 처리기 등)들의 실행 시간이 결정적(deterministic)[13]이고 조인 어트리뷰트의 자료 분포가 균등분포를 따를 때는 [10,11]의 실행 동기화 개념이 질의 실행 때에도 유효하나, 실제 데이타베이스에서는 균등분포를 따르지 않는다[13]. [10]에서 사용한 데이타베이스 카탈로그의 기본 자료(릴레이션의 투플 수, 어트리뷰트의 수, 자료의 균등분포 가정)는 질의 최적화기가 사용할 수 있는 정적인 자료인데 이 자료들을 토대로 한 예측 값은 부정확하다[1,2]. [10]에서는 균등분포를 가정하고 중간결과 투플 수를 예측[12]하여 누적 실행 비용을 계산한 후 프로세서를 할당하였다. 그러나 파이프라인 처리 환경에서는 기존 2-way 병렬 해시 조인이 다수 존재하므로 자료 불균등 분포에 대한 영향은 더욱 심화된다. 따라서 데이타베이스의 정적인 통계 자료에 없고, 자료 값에 내재하여 동적으로 발생하는 작업 부하 불균형은 조인 알고리즘이 그 대비책을 준비하고 있어야 한다.

3.2 자료 불균형 분포의 정의와 그 처리 방법

해시 기반 병렬 조인 알고리즘은 두 가지 중요한 문제점을 안고 있다[14,15]. 1) 버켓(bucket)은 하나의 릴레이션(투플들의 집합)의 부분 집합인데 조인 어트리뷰트 값의 어떤 특징에 따라 분리된 투플들의 집합이다. 버켓은 릴레이션 분할(수평분할)의 기본 단위이므로 메모리에 전체가 저장될 수 있어야 한다. 그러나 조인 어트리뷰트 값의 불균등 분포로 인해 오버플로우가 발생할 수 있으며 그 처리[1,4] 때문에 성능이 저하된다. 본 논문에서는 파이프라인 처리 특성상 테이블 구축용 릴레이션 전체가 메모리에 들어갈 수 있으므로 버켓 오버플로우는 발생하지 않는다고 가정한다. 2) 해시 기반 병렬 조인 알고리즘은 분할된 부 릴레이션의 투플 수가 균등하게 각 처리 노드에 분배되어야 하는데 조인 어트리뷰트 값의 불균등 분포 성격 때문에 작업 부하가 모든 처리 노드에 균등히 분배되지 않는다[13]. 본 논문은 이 문제를 다룬다. 자료 불균형 분포란 어떤 주어진 어트리뷰트에서 어떤 값들이 다른 값들에 비해 많이 나타나는 현상이며 자료 불균형 분포에 대한 영향은 명확히 분석해야 할 필요가 있다[13,14,15].

3.2.1 2-way 병렬 해시 조인

어떤 조인 연산 R ⋈ S에 5개의 프로세서가 할당되어 2-way 병렬 해시 조인한다고 하자. 피 연산자인 릴레이션 R과 S는 여러 처리 노드에 수평으로 분할 저장되어 각 처리 노드는 지역 디스크에 R_i와 $S_i(i=0,..,4)$를 갖고 있다고 하자. 어떤 프로세서$_i$가 R_i를 지역 디스크에서 읽어 분할할 때 조인 어트리뷰트의 자료 불균형 분포는 그림3과 같이 버켓의 크기가 달라지는 부 버켓 스큐(sub-bucket skew)를 일으킬 수 있다. 이 버켓들을 버켓 번호대로 각 프로세서에 배정하면 프로세서#0이 Bucket[0]을 할당받는다. 만약 모든 R_i가 그림3과 같은 자료 불균형 분포를 갖는다면 프로세서#2가 할당받는 분할(partition)은 $|P_2| \cong 5 \times 240$ 이 되어 프로세서#2의 부하가 가장 크게 된다. 분할은 조인 연산에 직접 사용되므로 버켓 재배치 단계에서 분할의 균형을 유지할 필

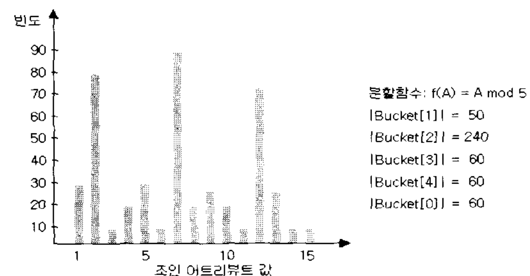


그림 3 자료 불균형 분포의 예

요가 있다[14,15]. 병렬 해시 조인 알고리즘의 재배치 단계에서 버켓을 각 프로세서에 할당하는데 분할 벡터(partitioning vector)를 사용한다. [14]에서는 릴레이션의 일부(샘플링)를 읽어 분할 벡터를 결정하였다.

또한 버켓의 수를 실제 존재하는 프로세서의 수보다 크게 하여(fine granularity) 분할의 불균등 현상을 줄이도록 하였다. [15]에서는 버켓의 크기를 충분히 크게 하여 피연산자 릴레이션 전체를 버켓들로 해시하고 버켓 크기의 전역 분포도를 알아낸 후에 병렬 해시 조인 알고리즘의 조인 단계 전에 분할의 균형을 유지하였다.

2-way 병렬 해시 조인 R ⋈ S에서 조인 어트리뷰트의 자료 불균형이 한쪽 릴레이션 또는 양쪽 릴레이션에 존재하는가에 따라 단일스큐와 이중스큐로 구분한다[13,14,15]. 그리고 2-way 조인에 참여하는 각 프로세서의 결과 릴레이션의 투플 수가 불균등 분포를 갖는 것을 조인스큐라고 한다[14]. 예를 들어 R의 조인 어트리뷰트 값이 {1, 1, 1, 1, 1, 2, 3} 이고 S의 조인 어트리뷰트 값이 {1, 2, 3, 4, 5, 6} 이면 단일스큐가 존재한다. 만약 S의 조인 어트리뷰트 값이 {1, 1, 1, 1, 2, 3} 이면 이중스큐가 존재한다. 이중스큐의 경우는 조인스큐를 수반한다.

2-way 병렬 해시 조인에서 분할의 균형을 유지하기 위해 R의 분할, S의 분할을 각각 또는 모두 고려할 수 있다. 한쪽의 분할만을 고려하는 경우는 테이블 구축용 릴레이션 분할이 더 우선한다[14].

3.2.2 다중 병렬 해시 조인의 파이프라인 처리

2-way 병렬 해시 조인[15]에서는 테이블 구축용 릴레이션과 검사용 릴레이션 모두의 부하 균형을 유지하도록 했다. 그런데 파이프라인 세그먼트는 테이블 구축용 릴레이션이 모두 메모리에 적재되어야 하는 특성이 있고, 투플 검사용 릴레이션은 일단 파이프라인이 실행되어야 계산되어진다는 특징으로 R과 S를 모두 고려한 분할의 균형유지가 제한적이다.

본 논문에서는 테이블 구축용 릴레이션에 대해서만 분할을 균등하게 유지하도록 하였다. 부하 균형 유지의 기본 단위는 우향트리인 파이프라인 세그먼트로 하여,

```
알고리즘 SPPJ(RDT)
입력            RDT: 다중 조인될 우향 이진 트리 (Right Deep Tree)
출력            릴레이션
BEGIN
1) // Table-building Phase
2)    // for all PNs in the system - Split Phase : split and redistribute building relations
3)       Each PN reads its portion of Ri from local disks to its local memory pages; // Ri is the building relation
4)       Each PN builds a large number of local buckets from the memory pages using a partition function;
5)       Each PN sends the buckets to their destinations in the same stage according to a round-robin partition vector;
6)    // for all PNs in the system - Build Phase : build local hash sub-tables
7)       Each PN receives its assigned buckets determined by the round-robin partition vector from other PNs;
8)       Each PN builds its local hash sub-table;
9) // Tuple-probing Phase : Pipeline Processing
10)   // for all PNs in the system
11)      Each PN reads its portion of S from disks to local memory pages; // S is the probing relation
12)       Each PN builds local S-buckets from the memory pages using a partition function;
13)       Each PN sends the S-buckets to their destination in the first stage according to the first stage's partition vector;
14)   // for all PNs in the first stage
15)      Each PN receives its matching S buckets of the local hash sub-table from other PNs in the system;
16)      Each PN probes tuples in the S-buckets against its local hash sub-table;
17)      If there are matches, the intermediate result tuples are generated, /* Join */
           and the result tuples are partitioned according to the next stage's partition function,
                  and sent to the destination nodes according to the next stage's partition vector; /* Split */
18)   // for all PNs in the middle stages // except the first and last stage.
19)      Each PN receives its matching S-buckets of the local hash sub-table from PNs in the previous stage;
20)      Each PN probes tuples in the S-buckets against its local hash sub-table;
21)      If there are matches, the intermediate result tuples are generated, /* Join */
           and the result tuples are partitioned according to the next stage's partition function,
                  and sent to the destination nodes according to the next stage's partition vector; /* Split */
22)   // for all PNs in the last stage
23)      Each PN receives its matching S-buckets of the local hash sub table from PNs in the previous stage;
24)      Each PN probes tuples in the S-buckets against its local hash sub table;
25)      If there are matches, the final result tuples are generated, /* Join */
           and the result tuples are written to local disks;
END
```

그림 4 정적 분할 알고리즘 (SPPJ: Static Partioning Pipelined hash Join)

```
알고리즘 ABPJ(RDT)
입력          RDT: 다중 조인될 우향 이진 트리 (Right Deep Tree)
출력          릴레이션
BEGIN
1) // Table-building Phase
2)     // for all PNs in the system - Split Phase : split, no redistribution
3)         Each PN reads its portion of Ri from local disks to its local memory pages; // Ri is the building relation
4)         Each PN builds a large number of local buckets from the memory pages using a partition function;
5)     // for all PNs in the stsem - Partition tuning Phase
6)         Each PN reports the size of its local buckets to the designated coordinating PN;//there are k coordinator(k:stage#);
7)         The coordinator collects the bucket distribution information and make a partition vector
               using the largest local bucket retaining strategy and best-fit decreasing strategy;
8)         The coordinator broadcast the partition vector and Each PN receives the partition vector;
9)     // for all PNs in the system - Bucket distribution Phase
10)        Each PN sends the local buckets to their destinations in the same stage
               according to the partition vector;
11)    // for all PNs in the system - Build local hash sub-tables
12)        Each PN receives its assigned buckets determined by the partition vector from other PNs;
13)        Each PN builds its local hash sub-table;
14) // Tuple-probing Phase : Pipeline Processing
15)    // Same as the SPPJ method
END
```

그림 5 적응적 분할 알고리즘 (ABPJ: Adaptive load Balancing Pipelined hash Join)

질의 실행 트리가 세그먼트로 구분된 우향트리[9] 그리고 부시트리[10,11]에 모두 적용할 수 있도록 하였다. 3.1절에서 설명한 기존의 파이프라인 해시 조인 알고리즘에 2-way 병렬 해시 조인의 분할 유지 기법[14,15]을 적용하여 두 가지 부하 균형 유지 파이프라인 해시 조인 알고리즘을 제안한다. 미리 고정된 분할 벡터에 의하여 부하를 분할하는 방법은 그림4와 같고, 자료의 분포도에 따라 능동적으로 분할하여 분할의 균형을 유지하는 방법은 그림5와 같다.

그림4의 정적 분할 파이프라인 해시 조인(SPPJ)은 3.1절에서 설명한 기존의 파이프라인 해시 조인 방법[9]에 부하균형 유지 요소를 추가하여 확장한 방법이다. 즉 분할 버켓의 개수를 실제 할당될 프로세서의 수보다 크게 하였고, Round-robin 으로 할당하였다. 기존 파이프라인 해시 조인은 분할 버켓의 수가 스테이지의 참여 프로세서 수와 같은 경우이다(이하 SPPJ(1) 이라 함). 그림4를 보면, 테이블 구축 단계의 릴레이션 분리(split) 단계 전에 미리 분할 벡터가 결정되어 있다. 릴레이션 분리 과정에서 만든 버켓을 분할 벡터로 할당하는 방법은 Round-robin 방식이다. 분할 벡터가 미리 결정되어 있으므로 릴레이션 분리 및 재배치(라인 2-5)와 해시 부 테이블 구축(라인 6-8)이 동시에 이루어질 수 있다. SPPJ의 비용 분석은 4.1.1절에서 자세히 설명한다. 릴레이션 분리(라인 4) 때 사용하는 버켓의 개수에 따른 성능의 변화는 5장 실험 결과에서 설명한다.

정적 분할 파이프라인 해시 조인(SPPJ)과 적응적 부하 균형 유지 파이프라인 해시 조인(ABPJ)의 차이점은 분할 벡터의 결정 시기와 결정 방법이다. 적응적 부하 균형 유지 파이프라인 해시 조인(ABPJ)은 테이블 구축 단계에서 릴레이션 분리(그림5의 라인 2-4)와 분할 벡터의 결정(그림5의 라인 5-8)이 엄격히 구분되어 있다. 분할 벡터의 결정은 SPPJ와는 달리 버켓의 자료 분포도에 따라 2장에서 설명한 [15]의 "Largest sub-bucket retaining strategy"와 "Best-fit decreasing strategy"에 의해 분할 벡터를 결정한다. 분할 벡터가 결정되면 릴레이션 재배치(그림5의 9-10)와 해시 부 테이블 구축(그림5의 라인 11-13)이 동시에 이루어진다. 분할 조율과정으로 수반되는 ABPJ의 오버헤드와 비용 분석은 4.1.2절에서 설명한다.

해시 테이블 구축 단계가 종료하면 SPPJ와 ABPJ 모두 동일한 투플 검사 단계를 갖는다. SPPJ 방법에서는 검사용 투플의 흐름이 스테이지별로 미리 고정된 분할 벡터에 의해 결정되며, ABPJ 방법은 조인할 한쪽 릴레이션의 버켓의 자료 분포도에 따라 균형을 유지한 새로운 분할 벡터에 의해 결정된다.

4장에서는 파이프라인 세그먼트의 실행모형, 비용모형 그리고 시뮬레이터의 입력이 되는 부하모형에 대하여 설명하고 SPPJ와 ABPJ의 성능 비교는 5장에서 한다.

4. 모의 실험

본 장에서는 조인 어트리뷰트의 자료 불균형 영향을 분석하고 부하 균형 유지 기능을 갖춘 SPPJ와 ABPJ

표 1 비용분석에 사용된 시스템 파라미터

파라미터	의 미	값
n_p	하나의 파이프라인 세그먼트에 할당된 프로세서 수	3~64개
M	프로세서의 메모리 크기	$\vert R\vert$
p_{speed}	프로세서의 명령어 처리율	25MIPS
t_{read}	디스크로부터 하나의 100byte 튜플을 읽는 CPU 비용	50μsec
t_{part}	하나의 튜플에 분할함수를 적용하는 CPU 비용	4μsec
t_{send}	하나의 튜플을 네트워크에 보내는 CPU 비용	4μsec
t_{rec}	네트워크로부터 하나의 튜플을 받는 CPU 비용	2μsec
t_{hash}	하나의 튜플에 해시함수를 적용하는 CPU 비용	4μsec
t_{insert}	하나의 튜플을 해시 부 테이블에 삽입하는 CPU 비용	20μsec
t_{comp}	하나의 튜플과 버켓내 엔트리와 비교하는 CPU 비용	2μsec
t_{build}	하나의 결과 튜플을 만드는 CPU 비용	40μsec
t_{write}	하나의 튜플을 디스크에 쓰는 CPU 비용	80μsec
D_{net}	하나의 p_{size}(4kbytes) 패킷의 전송 서비스 비용	1msec
D_{io}	하나의 d_{size}(4kbytes) 페이지의 전송 서비스 비용	20msec
$n_{buckets}$	SPPJ와 ABPJ의 분할에 사용하는 버켓의 개수	1~3092

방법을 비교 평가하기 위하여 개발한 모의 실험 환경에 대하여 설명한다.

4.1 실행 모형

3장에서 설명한 SPPJ 방법과 ABPJ 방법의 시뮬레이터를 C 언어를 사용하여 HP N4000 시스템에서 구현하였다. 성능비교의 척도는 하나의 파이프라인 세그먼트 실행시간으로 한다.

비용함수는 [9]에서 사용한 파이프라인 실행 모형의 비용함수를 기반으로 [15]의 작업분할에 관련된 오버헤드, 디스크 서비스 그리고 통신 서비스 비용을 추가하여 개발하였다.

모형의 파라미터 중 시스템 특성은 표1과 같이 [9]의 파라미터를 그대로 사용하여 자료 불균형의 영향을 객관적으로 판단할 수 있도록 하였다. 하나의 처리 노드는 디스크 서비스, CPU 처리, 프로세서 간 통신을 동시에 할 수 있다고 가정한다. 따라서 어떤 프로세서에 주어진 작업(투플 스트림)의 처리 시간은 투플들의 흐름을 처리하는데 소요된 시간으로 $max(T_{disk_delay}, T_{cpu_process}, T_{network_delay})$이다.

4.1.1 정적 분할 파이프라인 해시 조인의 비용 함수

3장에서 설명한 파이프라인 세그먼트의 각 스테이지$_i$에 피 연산자로서 테이블 구축용 릴레이션 R_i가 있고, R_i는 n_i개의 처리 노드의 지역 디스크에 균등한 크기로 분산되어 있다고 가정한다. 최초 테이블 검사용 릴레이션(S)은 파이프라인 세그먼트의 실행에 참여하는 N개의 처리 노드의 지역 디스크에 분산 저장되어 있다고 가정한다[9].

정적 분할 파이프라인 해시 조인 방법(SPPJ)은 테이블 구축 단계와 투플 검사 단계로 구분되어 실행된다.

$$T_{SPPJ} = T_{table_building} + T_{tuple_probing} \quad (1)$$

(1)식의 테이블 구축 비용, $T_{table_building}$에서 각 스테이지의 참여 프로세서는 독립적으로 스테이지의 테이블 구축용 릴레이션 R에 대한 부 릴레이션을 분할하여 메모리에 해시 부 테이블을 구축한다.

$$T_{table_building} = max\,\forall i(T_{table_building_stagei})$$
$$(i=1,..,k, \text{ 여기서 k는 스테이지 수}) \quad (2)$$

(2)식에서 $max(T_{phase_1},...,T_{phase_m})$는 독립적 처리과정 $_{phase_1,...,\ phase_m}$ 중 실행 비용이 가장 큰 값을 의미한다. 그러므로 테이블 구축 실행 시간이 최대인 스테이지가 전체 테이블 구축 비용을 결정한다.

$$T_{table_building_stagei}=max\,\forall j(T_{table_building_pnj})\ (j=1,..,n_i)$$
$$T_{table_building_pnj}=overlap(T_{split\ partition_phasej},\ T_{build_phasej}) \quad (3)$$

(3)식에서 $max(T_{phase_1}, T_{phase_2}, ..., T_{phase_m}) \leq overlap(T_{phase_1}, T_{phase_2}, ..., T_{phase_m}) \leq T_{phase_1} + T_{phase_2} + ... + T_{phase_m}$ 이며 실행의 겹침을 고려한다. SPPJ방법은 테이블 구축 때 분할 과정과 해시테이블 생성 과정이 동시에 진행한다.

$$T_{split\text{-}partition_phasej} = overlap(D_{io} \cdot \lceil |B_j|/d_{size} \rceil,\ t_{read} \cdot |B_j| + t_{part} \cdot |B_j| + t_{send} \cdot |B_j - P_j| + t_{rec} \cdot |P_j - B_j|,\ overlap(D_{net} \cdot \lceil |B_{hj}|/p_{size} \rceil,\ h=1,..,ni,\ h \neq j)) \quad (4)$$

(4)식에서 B_j는 분할(해시)함수에 의해 프로세서$_j$가 만드는 버켓 집합으로서 $|B_j| = |R_i|/n_i$이다. P_j는 재배치(redistribution) 후 프로세서$_j$가 해시 부 테이블의 입력으로 직접 사용하는 버켓 집합이다. B_{hj}는 같은 스테이지 소속의 다른 프로세서$_h$가 분할한 릴레이션의 버켓 집합 중 분할벡터에 의하여 프로세서$_j$의 분할로 배정되는 버켓 집합이다. 스테이지에$_i$ 할당된 각 처리 프로세서$_j$는 내부릴레이션 R_i의 부 릴레이션(크기$|R_i|/n_i$)을 읽고(t_{read}) 부 버켓들로 분할하여(t_{part}) 버켓 집합 B_j를 만든다. 미리 정의된 분할 벡터에 의하여 스테이지$_i$ 소속 다른 프로세서가 사용할 버켓 집합들(B_j-P_j)은 고속 망을 통하여 다른 프로세서에게 보내고 다른 처리 노드들이 보내는 버켓 집합($P_j-B_j = \sum h=1,..,n_i|B_{hj}|, h \neq j$)은 받는다. 조인 어트리뷰트 값이 균등분포인 경우 각 프로세서가 다른 프로세서에게 보내거나 받는 버켓들의 크기는 $(n_i-1/n_i)|R_i|/n_i$이다. [9,10]에서는 조인 어트리뷰트 값이 균등 분포인 경우만 고려하였으므로, $|P_j| = |R_i|/n_i$이다.

$$T_{build_phasej} = (t_{hash} + t_{insert}) \cdot |P_j|$$

다음에 투플 검사의 실행 비용($T_{tuple_probing}$)을 알아보자. 테이블 구축 단계에서는 각 스테이지의 테이블 구축이 스테이지별로 독립적으로 수행된다. 그러나 투플 검사는 파이프라인 데이타 흐름에 참여하는 스테이지들이 모두 연결되어 수행된다. 그리고 스테이지 내부에는 참여 프로세서가 스테이지 내부 데이타 흐름에 모두 참여한다.

$$T_{tuple_probing} = T_{pipeline_setup} + T_{pipeline_progress} + T_{pipeline_depletion} \quad (5)$$

(5)식의 투플 검사 비용은 세 가지 단계로 구분된다. 첫 번째는 파이프라인 설정 비용, 두 번째는 안정적 처리 단계 비용, 그리고 세 번째는 파이프라인 마무리 비용이다. 파이프라인 설정 비용은 검사용 릴레이션 투플 흐름이 처음 스테이지에서 마지막 스테이지까지 최초로 도달하는데 소요되는 비용이다. 파이프라인 마무리 비용은 파이프라인이 비워지는데 소요되는 비용이다. 릴레이션이 큰 경우는 투플 검사의 대부분 시간이 안정적 처리 단계에서 소비되므로 (6)식과 같이 투플 검사 시간을 파이프라인의 안정적 처리 시간으로 근사 시킨다. 파이프라인 실행의 전 과정과 Burst effect[9]는 모의 실험을 통해 구현하였다.

$$T_{tuple_probing} \approx T_{pipeline_progress} \approx \max \forall i(T_{pipeline_progress_stagei}) \ (i=1,..,k) \quad (6)$$

$$T_{pipeline_progress_stagei} = \max \forall j(T_{pipeline_progress_pnj}) \ (j=1,..,n_i)$$

$$T_{pipeline_progress_pnj} = overlap(T_{pre_joinj}, T_{joinj}, T_{post_joinj}) \quad (7)$$

(7)식의 구성 비용은 프로세서가 소속된 스테이지의 위치에 따라 다르며 다음과 같다:

프로세서$_j$가 첫 번째 스테이지의 구성원인 경우:

$$T_{pre_joinj} = overlap(D_{io} \cdot \lceil |B_j|/d_{size} \rceil, t_{read} \cdot |B_j| + t_{part} \cdot |B_j| + t_{send} \cdot |B_j - Q_j| + t_{rec} \cdot |Q_j - B_j|, overlap(D_{net} \cdot \lceil |B_{hj}|/p_{size} \rceil, \text{여기서 } h=1,..,N, h \neq j)) \quad (8)$$

(8)식에서 B_j는 최초 검사용 릴레이션 S의 부 릴레이션(크기:$|S|/N$)에 대해 프로세서$_j$가 분할(해시)함수로 만든 버켓 집합이다. Q_j는 첫 번째 스테이지에 입력되는 검사용 릴레이션 S의 부분 집합으로 테이블 구축 단계에서 고정된 분할 벡터에 의해 프로세서$_j$에 할당된 검사용 릴레이션 분할(partition)이다. 자료 불균형이 없는 경우 $|Q_j| = |S|/n_1$이다.

$$T_{joinj} = (t_{hash} + t_{comp}) \cdot |Q_j| + t_{build} \cdot |I_{1j}|$$

$$T_{post_joinj} = (t_{part} + t_{send}) \cdot |I_{1j}| \quad (9)$$

(9)식에서 I_{1j}는 첫 번째 스테이지가 만든 중간 결과 릴레이션 I_1 중 프로세서$_j$가 만든 릴레이션이다. 스테이지의 조인 선택도가 일정하다고 가정할 때 $|I_1|$은 자료 불균형과 관계없이 동일한 크기이지만 자료 불균형에 의해 피 연산자인 내부 릴레이션 $|P_j|$와 외부 릴레이션 $|Q_j|$가 참여 프로세서마다 균등하지 않으면 $|I_{1j}| \neq |I_1| / n_1$이 되어 프로세서마다 조인 선택도가 다르게 되므로 조인스큐[14]를 일으킨다. 중간 결과 릴레이션 I_1는 다음 스테이지의 입력이 된다.

프로세서$_j$가 중간 스테이지의 구성원인 경우:

$$T_{pre_joinj} = overlap(D_{io} \cdot \lceil |B_j|/d_{size} \rceil, t_{read} \cdot |B_j| + t_{part} \cdot |B_j| + t_{send} \cdot |B_j| + t_{rec} \cdot |Q_j|, overlap(D_{net} \cdot \lceil |Q_{hj}|/p_{size} \rceil, \text{여기서 } h=1,..,n_{i-1})) \quad (10)$$

(10)식에서 B_j는 최초 검사용 릴레이션 S의 부 릴레이션(크기:$|S|/N$)에 대해 프로세서$_j$가 분할 함수로 만든 버켓 집합이다. Q_j는 이전 스테이지의 중간 결과 릴레이션 I_{i-1}중 프로세서$_j$에게 입력되는 I_{i-1}의 부분 집합이다. I_{i-1}의 분할은 테이블 구축 단계에서 고정된 스테이지$_i$의 분할 벡터에 의하여 결정된다. Q_{hj}는 이전 스테이지$_{i-1}$의 구성 프로세서h($h=1..n_{i-1}$)가 스테이지$_i$의 프로세서$_j$에게 보내는 릴레이션으로서 I_{i-1}의 부분 집합이다.

$$T_{joinj} = (t_{hash} + t_{comp}) \cdot |Q_j| + t_{build} \cdot |I_{ij}|$$

$$T_{post_joinj} = (t_{part} + t_{send}) \cdot |I_{ij}|$$

프로세서$_j$가 마지막 스테이지의 구성원인 경우:

T_{pre_joinj}와 T_{joinj}은 프로세서$_j$가 중간 스테이지 구성원인 경우와 같다.

$$T_{post_joinj} = overlap(t_{write} \cdot |I_{kj}|, D_{io} \cdot \lceil |I_{kj}|/d_{size} \rceil)$$

4.1.2 적응적 부하 균형 유지 파이프라인 해시 조인의 비용 함수

적응적 부하 균형 유지 파이프라인 해시 조인 방법(ABPJ)은 테이블 구축 단계와 투플 검사 단계로 구분되어 실행된다.

$$T_{ABPJ} = T_{table_building} + T_{tuple_probing}$$

투플 검사 단계의 비용, $T_{tuple_probing}$의 계산식은 SPPJ와 동일하므로 설명을 생략한다.

$$T_{table_building} = \max \forall i(T_{table_building_stagei})\ (i=1,..,k,\ \text{여기서 k는 스테이지 수})$$

$$T_{table_building_stagei} = \max \forall j(T_{table_building_pnj})\ (j=1,..,ni)$$

$$T_{table_building\ pnj} = T_{split_phasej} + overlap(T_{partition_phasej}, T_{build_phasej}) \quad (11)$$

SPPJ와는 달리 (11)식에서 릴레이션 분리(split) 단계와 분할(partition) 단계가 구분되고 해시테이블 생성은 분할 단계와 동시에 진행한다.

$$T_{split_phasej} = overlap(D_{io} \cdot \lceil |B_j|/d_{size} \rceil, (t_{read} + t_{part}) \cdot |B_j|)$$
$$T_{partition_phasej} = overlap(t_{send} \cdot |B_j - P_j| + t_{rec} \cdot |P_j - B_j|, overlap(D_{net} \cdot \lceil |B_{hj}|/p_{size} \rceil, \text{여기서 } h=1,..,n_i, h \neq j)) \quad (12)$$

(12)식에서 B_j는 프로세서$_j$가 분할 함수로 만드는 버켓 집합으로서 $|B_j| = |R_i|/n_i$이다. P_j는 재배치 후 프로세서$_j$가 직접 해시 부 테이블의 입력으로 사용하는 버켓 집합이다. B_{hj}는 같은 스테이지 소속의 다른 프로세서$_h$가 분할한 릴레이션의 버켓 집합 중 분할 벡터에 의하여 프로세서$_j$의 분할로 배정되는 버켓 집합이다.

$$T_{build_phasej} = (t_{hash} + t_{insert}) \cdot |P_j|$$

4.2 작업 부하 모형

SPPJ와 ABPJ 방법을 구현한 시뮬레이터의 입력은 하나의 파이프라인 세그먼트인 우향 이진 실행 트리이다. 시뮬레이터의 작업부하 파라미터는 표2와 같다. 릴레이션의 조인 어트리뷰트의 자료 불균형을 모형화 하기 위해 자료 불균형(data skew) 모형은 Zipf 분포[13,15]를 사용하였다. 다중 조인 $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$ ($m > 2$) 에서 임의의 릴레이션 R을 분리(split)하여 버켓 집합 $B = \{B_1, B_2, B_3, \ldots, B_s\}$ 일 때, B_i 에 속하는 투플들의 개수는 다음 식에 의해 결정된다[15]:

$$|B_i| = |R| / (i^{Zb} \sum_{j=1,\ldots,s}(1/j^{Zb}))$$

위식에서 $0 \le Zb \le 1$ 이며, $Zb = 0$인 경우 균등분포가 되고, $Zb = 1$이면 Zipf 분포가 된다. Zb를 버켓 스큐도라 부른다.

실험에 사용된 우향 이진 트리는 3-way ~ 9-way 조인까지 구성하였다. m-way 조인에서 각각의 2-way 조인에 사용된 조인 어트리뷰트는 다른 2-way 조인과 서로 독립적이라고 가정하였다. 모든 중간 결과 릴레이션의 투플 크기도 표2와 같이 고정시켰다. 질의 실행 트리에서 모든 2-way 조인의 릴레이션 크기는 같다고 가정하였다. 질의 실행 트리에서 모든 2-way 조인의 릴레이션 카티널리티를 고정시키면 릴레이션의 조인 순서나 프로세서의 할당 방법[10]에 영향을 받지 않고 자료 불균형 분포의 영향에만 초점을 맞출 수 있다. 이러한 우향 이진 트리들에 대하여 버켓 스큐도, 릴레이션 크기, 조인 선택도, 그리고 프로세서의 개수를 변화시켜 실험하였다. 5장에서 실험 결과를 설명한다.

표 2 비용분석에 사용된 작업부하 파라미터

파라미터	의 미	값
실행트리	질의 처리기의 입력으로서 우향 이진 트리	3 ~ 9 way
js	2-way 조인의 조인 선택도(join selectivity)	1e-7 ~ 4e-6
|R|	테이블 구축용 릴레이션의 크기(투플 수)	50만 ~ 200만
|S|	테이블 검사용 릴레이션의 크기(투플 수)	50만 ~ 200만
|Tuple|	하나의 투플 크기	100bytes
Zb	자료 불균형도	0.0 ~ 1.0

5. 실험 결과 분석

본 장에서는 4장에서 개발한 모의 실험 환경으로 실험한 결과에 대하여 설명한다. 파이프라인 다중 해시 조인에서 자료 불균형 분포가 성능에 주는 민감도를 분석하고, 본 논문에서 제안한 SPPJ 방법과 ABPJ 방법의 성능을 비교 분석한다.

자료 불균형에 대한 실험 공간은 표3과 같다. 관심의 대상인 Case_S_U와 Case_S_S에 대해서 중점적으로 설명한다. Case_U_U와 Case_U_S의 경우는 SPPJ와 ABPJ에 의해 영향을 받지 않으므로 실험결과를 제시하지 않았다. 스테이지(2-way 병렬 해시 조인) 간 존재할 수 있는 자료 불균형은 최악의 경우에 대해서 실험을 하였다. 즉 이전 스테이지의 조인 스큐로 중간 결과를 가장 많이 생성하는 프로세서가 다음 스테이지의 참여 프로세서 중 해시 테이블이 가장 크게 생성된 프로세서

표 3 2-way 조인에서 자료 불균형의 종류

		테이블 검사용 릴레이션(S)	
		균등 분포 (Uniform)	불균형 분포 (Skewed)
테이블 구축용 릴레이션(R)	균등분포 (Uniform)	Case_U_U	단일스큐 Case_U_S
	불균등 분포 (Skewed)	**단일스큐 Case_S_U**	**이중스큐 Case_S_S**

로 자료를 전송하도록 하였다.

그림6은 3.1절에서 설명한 기존의 파이프라인 해시 조인의 자료 불균형에 대한 실행 비용을 나타낸다. 기존 방법(SPPJ(1))은 가장 단순한 분할 방법을 사용한 경우로서, SPPJ 방법에서 버켓의 개수가 분할될 프로세서의 수와 같은 경우이다.

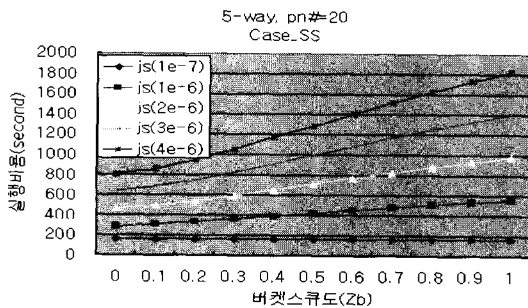


그림 6 자료 불균형에 따른 기존 방법의 실행비용 증가

그림6은 5개의 스테이지 각각에 4개씩 전체 20개의 프로세서가 할당된 경우 실행 비용이다. 첫 번째 2-way 조인의 조인 선택도를 10^{-7} ~ 4×10^{-6} 으로 변화시키고, 나머지 2-way 조인의 조인 선택도는 10^{-6}으로 고정시켰다. Zb = 0인 경우는 균등 분포로서 참여 프로세서 모두가 균등한 작업 부하를 갖는 경우이다. 결과 릴레이션이 10만개(js:1e-7)인 경우에는 자료 분포에 관계없이 일정한 실행 시간을 갖는다. 그러나 조인 선택도가 커지면서 결과 릴레이션이 백만개(js:1e-6)에서 4백만개(js:4e-6)로 늘어나는데, 자료의 불균형도가 심해질수록 실행시간 증가도가 커지는 것을 알 수 있다. 자료 불균형 분포는 투플 검사 단계에서 스테이지 별로 특정 프로세서에 많은 투플을 입력시키며 이 프로세서는 많은 중간 결과 투플을 생성하여 상위 스테이지의 프로세서들에게 전달하게 되는데, 입력 자료량이 커질수록 불균

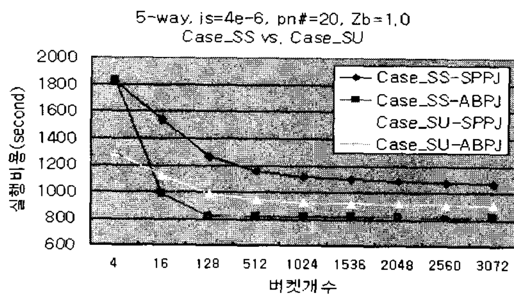


그림 7 버켓 크기에 따른 실행 비용의 감소

형도(Zb)에 의한 작업량의 상대적 불균형이 심해지기 때문에 실행 시간 그래프의 경사도가 커진다.

그림6의 기존 파이프라인 해시 조인의 자료 불균형 영향을 줄이기 위해 버켓의 크기를 증가시키며 SPPJ 방법과 ABPJ 방법을 실행한 결과를 그림7에 나타낸다. 그림7은 그림6에서 자료 불균형의 영향이 가장 심한 js=4e10-6, Zb=1.0에 대한 실험 결과이다. [14,15]의 경우처럼 물리적 프로세서의 수보다 큰 수의 버켓을 가지고 분할을 해야 좀 더 균형 있는 작업의 분할이 가능함을 알 수 있다. 그림7을 보면 버켓의 개수를 적게 사용한 경우에도 작업 부하의 균형 유지 능력은 ABPJ가 더 우수함을 알 수 있다.

그림8은 파이프라인 실행에서 입력량이 충분히 많고[7], 각 스테이지의 조인 선택도와 자료 불균형도가 일정하면, 스테이지의 개수(질의실행 트리의 높이)에 관계없이 일정한 실행

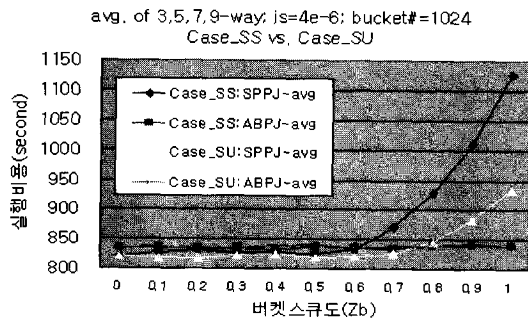


그림 8 m-way조인에서 자료 불균형에 따른 실행 비용

시간을 갖는다는 것을 보여주는 예이다(4.1.1의 비용식(6))[9]. 릴레이션 크기와 조인 선택도는 그림7과 같고, 스테이지 각각에 4개씩 프로세서를 할당시켜 3-way부터 9-way까지 조인 결과의 평균값을 나타내었는데, 3.2.2절에서 설명한 ABPJ의 오버헤드 때문에 SPPJ는 자료 불균형도가 작은 경우에는 유리하다. 그러나 자료 불균형도가 0.5보다 커지면서 실행 비용이 증가한다. 물론 SPPJ의 실행 시간 증가도는 그림7의 버켓 크기가 증가할수록 감소하겠지만 ABPJ는 이미 SPPJ의 실행 시간 최저치에 도달해 있다. 여기서부터 실험에 사용한 버켓의 개수는 1024개로 하였는데, 이것은 모의 실험의 실행 시간을 고려한 개수이고, 최적 버켓 수[15]는 아니다. 그림7을 보면 버켓의 개수가 3072개인 경우도 상대적인 실행 차이는 같다. 최적 버켓 수는 데이타베이스 버퍼 관리 문제[1,2,15]로서 본 논문에서는 고려하지 않

았다.

그림9는 5-way조인에서 모든 2-way 조인 선택도를 js=1e10-6로 고정시키고 스테이지의 각 릴레이션 크기를 균등히 증가시킨 경우에 실행 비용의 증가를 나타낸다. Zb=0~1의 평균값(avg)과 Zb=1(worst)을 나타내었다. 릴레이션의 크기가 80만 이내의 경우에 표1의 파라미터에서는 입력 제한 모드[7]로 파이프라인이 처리되기 때문에, 기존 방법(SPPJ(1))과 본 논문에서 제안한 SPPJ와 ABPJ 모두 실행 시간에 큰 차이를 보이지 않는다. 그러나 릴레이션 크기가 80만 이상이 되면, 처리기 제한 모드[7]로 동작되고, 최종 결과를 얻는 마지막 스테이지에서 많은 병목 현상을 일으키면서 불균형의 영향이 나타나기 시작한다. 결과 수집의 자료 불균형[14] 영향은 병렬처리기 구조나 프로세서 할당 방법에 따라 영향의 정도는 줄일 수 있지만, 프로세서 간 상대적인 차이를 줄이기 위해서는 부하 균형 유지 기법이 필요하다. ABPJ 방법은 기존 방법(SPPJ(1))의 실행속도를 50% 이상 감소시킨다.

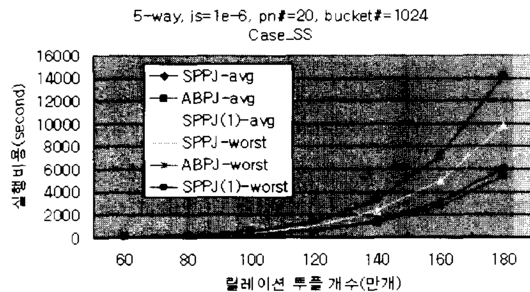


그림 9 릴레이션 크기에 따른 비용의 증가

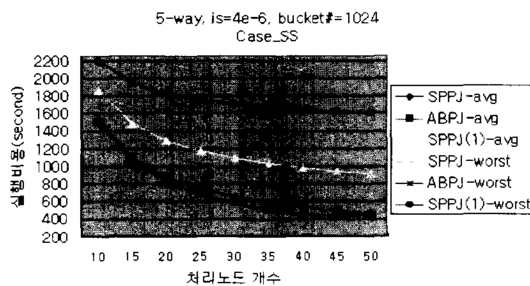


그림 10 처리 노드 증가에 따른 비용의 감소

그림10은 처리 노드의 증가에 따른 속도 개선 효과를 나타낸다. Zb=0~1의 평균값(avg)과 Zb=1(worst)을 나타내었다. 기존 방법(SPPJ(1))도 처리 노드 수가 증가하면서 전체 입출력 대역폭의 증가와 분할 버켓 수의 증가로 인하여 성능 개선의 효과가 있지만, 병목 현상으로 인하여 성능 개선은 매우 제한적이다. 본 논문에서 제안한 SPPJ와 ABPJ 방법은 조인 어트리뷰트의 자료 불균형을 대비한 효과적인 부하 균형 유지 능력을 갖고 있으므로, 노드 수의 증가에 따른 시스템 전체의 입출력 대역폭을 충분히 활용하고 있다.

6. 결 론

조인 어트리뷰트의 자료 불균형이 m-way 병렬 해시 조인의 파이프라인 실행에 주는 영향에 대하여 분석하고 부하 균형을 고려한 새로운 알고리즘을 제안하였다. 파이프라인 실행은 모든 테이블 구축용 릴레이션이 메모리에 적재되어야 하고, 투플 검사용 릴레이션의 자료 불균형도는 파이프라인 실행이 진행되어야 알 수 있다. 이러한 파이프라인 실행의 특징을 고려하여 테이블 구축용 릴레이션의 적재 부하를 Round-robin 방식으로 정적 분할하는 SPPJ 방법과 자료 분포도를 통해 동적 분할하는 ABPJ 방법을 제안하였다.

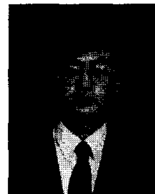
SPPJ(1)과 같은 단순한 분할 방법에 의한 기존의 파이프라인 병렬 해시 조인은 피 연산자 릴레이션들이 크거나 조인 선택도가 큰 경우에 조인 어트리뷰트의 자료 불균형으로 심각한 성능 저하를 가져올 수 있다. 따라서 자료 값에 내재하여 동적으로 발생하는 작업 부하 불균형은 조인 알고리즘이 그 대비책을 준비하고 있어야 한다. 본 논문에서 제안한 SPPJ 방법과 ABPJ 방법은 모두 부하 불균형 영향을 감소시키는 효과가 있다. ABPJ 방법은 동적 부하균형 유지의 오버헤드가 있으므로, 자료 불균형도가 작거나 릴레이션 크기나 조인 선택도가 작은 경우에는 SPPJ 방법이 유리하다. 그러나 그 반대의 경우는 ABPJ 방법이 더 효과적이다. 부시 실행 트리[11]의 경우 질의 최적화기가 만든 각 할당 노드들의 실행 동기화는 매우 중요한데, 예측이 어려운 조인 어트리뷰트의 자료 불균형 효과는 프로세서 할당을 어렵게 한다. SPPJ와 ABPJ 방법은 [10,11]의 균등 자료 분포 가정에 따른 파이프라인 세그먼트의 실행시간 예측을 지원한다.

앞으로는 본 논문의 테이블 구축용 릴레이션의 부하 균형 유지 방법을 확장하여 투플 검사용 릴레이션에 대한 자료 불균형을 함께 고려한 방법으로서 샘플링을 통한 파이프라인 재구축 방법이나 테이블 구축용 릴레이션의 해시 부 테이블을 스테이지의 참여 프로세서들에 중복하여 구축한 후 투플 검사용 릴레이션을 중복해서 병렬 해시 조인시키는 부분집합 복제[14]에 의한 방법

등을 연구할 필요가 있다.

참고문헌

[1] Priti Mishra, Margaret H. Eich, "Join Processing in Relational Databases," ACM Computing Surveys, vol. 24, no. 1, pp. 63-113, March 1992.
[2] Goetz Graefe, "Query Evaluation Techniques for Large Databases," ACM Computing Surveys, vol. 25, no. 2, pp.73-170, June 1993.
[3] 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.
[4] D.A. Schneider and D.J. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment," Proc. SIGMOD Conf., pp.110-121, May 1989.
[5] D. Schneider, D.J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multicomputer Database Machines," Proc. 16th Int'l Conf. VLDB, pp. 469-480, August 1990.
[6] N. Roussopoulos, H. Kang, "A Pipeline n-way Join Algorithm based on the 2-way Semijoin Program," IEEE Trans. Knowledge and Data Engineering, vol. 3, no. 4, pp.461-473, December, 1991.
[7] A. Wilschut and P. Apers, "Dataflow query execution in parallel main memory environment," Proc. First Conf. Parallel and Distributed Information Systems, pp.68-77, December 1991.
[8] Ming-Syan Chen, Hui-I Hsiao, Philip S. Yu, "Applying Hash Filters to Improving the Execution of Bushy Trees," Proc. 14th Int'l Conf. VLDB, pp.505-516, August 1993.
[9] Ming-Syan Chen, Mingling Lo, Philip S. Yu, Honesty C. Young, "Applying Segmented Right-Deep Trees to Pipelining Multiple Hash Joins," IEEE Trans. Knowledge and Data Engineering, vol. 7, no. 4, pp.656-668, August 1995.
[10] Hui-I Hsiao, Ming-Syan Chen, "Parallel Execution of Hash Joins in Parallel Databases," IEEE Trans. Parallel and Distributed Systems, vol. 8, no. 8, pp.872-883, August 1997.
[11] 이규옥, 홍만표, "페이지 실행시간 동기화를 이용한 다중 해쉬 결합에서 결합률에 따른 효율적인 프로세서 할당 기법," 정보과학회지논문지:시스템 및 이론, 제28권, 제3호, pp. 144-154, 2001.
[12] Ming-Syan Chen, Philip S. Yu, "Interleaving a Join Sequence with Semijoins in Distributed Query Processing," IEEE Trans. Parallel and Distributed Systems, vol. 3, no. 5, pp.611-621, September 1992.
[13] M.Seetha Lakshmi, Philip S. Yu, "Effectiveness of Parallel Joins," IEEE Trans. Knowledge and Data Engineering, vol. 2, no. 4, pp.410-424, December 1990.
[14] D.J. DeWitt, J.F. Naughton, D.A. Schneider, S. Seshadri, "Practical Skew Handling in Parallel-joins," Proc. Int'l Conf. VLDB, pp.27-40, August 1992.
[15] Kien A. Hua, Chiang Lee, Chau M. Hua, "Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning," IEEE Trans. Knowledge and Data Engineering, vol.7, no.6, pp. 968-983, December 1995.


문 진 규
1986년 충남대학교 계산통계학과 학사. 1990년 충남대학교 계산통계학과 석사. 2001년 충남대학교 컴퓨터과학과 박사수료. 1989년~현재 국방과학연구소 선임연구원. 관심분야는 데이타베이스, 정보통신 등

진 성 일
1978년 서울대학교 계산통계학과 학사. 1980년 한국과학기술원 전산학과 석사. 1994년 한국과학기술원 전산학과 박사. 1988년 ~ 1989년 미국 Northwestern대학 객원교수. 1990년 ~ 1992년 충남대학교 전자계산소 소장. 1996년 ~ 2000년 충남대학교 소프트웨어 연구센터 소장. 1980년 ~ 현재 충남대학교 정보통신공학부 교수. 관심분야는 데이터베이스, 멀티미디어 등

조 성 현
1978년 서울대학교 계산통계학과 학사. 1980년 서울대학교 대학원 계산통계학과 석사. 1980년 7월 ~ 1983년 6월 육사 교관. 1995년 UCLA 전산학과 박사. 1995년 9월 ~ 1996년 8월 UCLA 전산학과 Postdoctoral Fellow. 1996년 9월 ~ 현재 홍익대학교 교수. 현재 소프트웨어·게임학부장. 관심분야는 분산운영체제, 분산데이터베이스, 컴퓨터 및 네트워크 보안