데이타베이스 공유 시스템에서 동적 부하분산을 지원하는 해쉬 조인 알고리즘들의 성능 평가

문 애 경† · 조 행 래††

요 약

기존에 제안된 대부분의 병렬 조인 알고리즘들은 데이타베이스가 여러 처리 노드에 분할되어 저장되는 데이타베이스 분할 시스템을 가정하였다. 데이타베이스 분할 시스템은 다수의 노드들을 연결할 수 있으며 지리적으로 분산된 환경도 지원할 수 있다는 장점을 갖지만, 데이타베이스 공유 시스템에 비해 부하 분산이나 시스템 가용성이 떨어진다는 단점을 갖는다. 본 논문에서는 데이타베이스 공유 시스템의 특성을 이용한 동적 부하분산 기법을 제안하고, 제안한 동적 부하분산 기법을 이용하여 기존의 해쉬 조인 알고리즘들을 데이타베이스 공유 시스템에 확장한다. 그리고, 다양한 시스템 구성 및 데이타베이스 부하 환경에서 모의 실험을 수행함으로써 데이타베이스 공유 시스템에서 동적 부하분산 기법의 효과 및 해쉬 알고리즘들의 성능 차이를 정량적으로 분석한다.

Performance Evaluation of Hash Join Algorithms Supporting Dynamic Load Balancing for a Database Sharing System

Aekyung Moon† · Haengrae Cho††

ABSTRACT

Most of previous parallel join algorithms assume a database partition system(DPS), where each database partition is owned by a single processing node. While the DPS is novel in the sense that it can interconnect a large number of nodes and support a geographically distributed environment, it may suffer from poor facility for load balancing and system availability compared to the database sharing system(DSS). In this paper, we propose a dynamic load balancing strategy by exploiting the characteristics of the DSS, and then extend the conventional hash join algorithms to the DSS by using the dynamic load balancing strategy. With simulation studies under a wide variety of system configurations and database workloads, we analyze the effects of the dynamic load balancing strategy and differences in the performances of hash join algorithms in the DSS.

1. 서 론

저렴한 처리 노드들을 연동하여 고성능 트랜잭션 처리 및 복잡한 질의에 대한 빠른 응답 시간을 지원하기 위한 구조로 데이타베이스 분할 시스템(Database Partition System : DPS)과 데이타베이스 공유 시스템(Database Sharing System : DSS)이 제안되었다[1, 2, 13]. DPS는 전체 데이타베이스를 분할하여 각 노드가 데이타베이스의 일정 분할을 관리하는 구조이며, DSS는 모든 노드들이 전체 데이타베이스를 공유하는 구조를 갖는다. DPS는 다수의 노드들을 연결할 수 있으며 데이타베이스가 지리적으로 분산된 환경도 지원할 수 있다는 장점을 갖지만, 하나의 노드가 고장날 경우 그 노드가

† 준 회 원 : 영남대학교 컴퓨터공학과
†† 정 회 원 : 영남대학교 컴퓨터정보통신공학부 교수

논문접수 : 1999년 9월 7일, 심사완료 : 1999년 11월 8일

관리하고 있는 데이타베이스 분할은 사용할 수 없다는 단점을 갖는다. 뿐만 아니라, 노드들간의 부하 분산이 어렵고 확장성도 떨어진다.

본 논문에서는 DSS를 위한 병렬 조인 알고리즘을 제안하고 성능을 분석한다. 현재까지 제안된 대부분의 병렬 조인 알고리즘들은 DPS를 가정하였으며, DSS를 위한 병렬 조인 알고리즘들은 거의 제안되지 않았다[7, 16]. 물론 DPS에서 제안된 기본 개념들이 DSS에서도 적용될 수 있지만, DSS에서는 각 노드들이 전체 데이타베이스를 액세스할 수 있으므로 보다 효율적인 병렬 조인 알고리즘의 개발이 가능하다. 뿐만 아니라, DPS의 경우 다른 노드가 관리하는 데이타베이스 분할을 액세스하기 위해서는 노드들간의 메시지 패싱이 필요하므로, 병렬 조인 처리를 위하여 동적 부하분산 기법을 적용한다면 데이타 전송 오버헤드가 커진다. 따라서 대부분의 DPS에서는 정적 부하분산 기법이나 제한된 형태의 동적 부하분산 기법만이 사용된다[3, 4, 11, 12, 19]. 이에 비해 DSS는 모든 노드에서 데이타베이스를 공유하므로 데이타 전송이 필요 없고, 따라서 동적 부하분산 기법의 구현이 용이하며 DPS에 비해 병렬 질의 처리에 더 효과적이다[13, 14].

이러한 관점에서 본 논문에서는 먼저 DSS에서 "Data Skew"[20]와 같은 부하 불균형 현상이 발생할 경우, 부하가 집중된 노드의 작업을 다른 노드로 재할당하는 동적 부하분산 기법을 제안한다. 그리고, 기존에 제안된 해쉬 조인 알고리즘들에 대해 동적 부하분산 기법을 적용함으로써 DSS 환경으로 확장하였다. 해쉬 조인 알고리즘들은 조인할 릴레이션들을 해쉬 함수를 이용하여 몇 개의 버킷들로 분할하고, 각 버킷에 대한 조인 연산은 상호 독립적으로 실행할 수 있으므로 병렬 처리에 적합하다는 장점을 갖는다[4, 5, 7, 10]. 해쉬 조인 알고리즘들의 성능을 분석하기 위하여 DSS를 위한 성능 평가 모형을 개발하였으며, 다양한 시스템 구성 및 데이타베이스 부하 환경에서 모의 실험을 수행하였다. 모의 실험 결과를 이용하여 DSS에서 동적 부하분산의 효과 및 해쉬 조인 알고리즘들의 상대적인 성능 차이를 정량적으로 분석할 수 있다.

본 논문의 구성은 다음과 같다. 2절에서는 관련 연구를 살펴보고, 3절에서는 본 논문에서 제안한 DSS를 위한 동적 부하분산 기법 및 해쉬 조인 알고리즘들을 설명한다. 4절에서는 DSS 성능 평가 모형 및 성능 분석을 위해 채택된 매개 변수를 설명한다. 5절에서는 성능 평가 모형을 이용하여 실험한 결과들을 분석하고, 끝으로 6절에서 결론을 맺는다.

2. 관련 연구

2.1 해쉬 조인 알고리즘

해쉬 조인 알고리즘은 분할 단계와 조인 단계로 나누어진다[9, 18]. 조인 애트리뷰트가 각각 JA_R, JA_S인 릴레이션 R과 S를 조인할 경우, 분할 단계에서는 각각의 조인 애트리뷰트에 공통된 해쉬 함수를 적용하여 R과 S를 버킷이라 불리는 n개의 하위 집합, $R_1, R_2, \cdots, R_n$ 그리고 $S_1, S_2, \cdots, S_n$으로 분할한다. 조인 단계에서는 R의 버킷에 저장된 튜플들로 해쉬 테이블을 만들고, S의 버킷에 저장된 튜플들로 해쉬 테이블을 검색한다. 이때 조인 애트리뷰트의 값이 일치하는 튜플들이 조인 결과로 산출된다. 임의의 버킷에 속하는 튜플 $r \in R_i$과 $s \in S_j$에 대해, $i \neq j$이면 $r.JA_R \neq s.JA_S$이다. 즉, 각 버킷에는 조인 애트리뷰트에 대해 동일한 해싱 값을 가지는 튜플들만 저장되므로 버킷들에 대한 조인 연산은 상호 독립적이다. 따라서, 조인 연산을 버킷 단위로 여러 노드에 분산시켜 동시에 처리함으로써 병렬 조인 처리가 가능해진다. 조인의 결과는 식 (1)과 같다.

$$R \bowtie S = \bigcup_{i=1}^{i=n} R_i \bowtie S_i \quad (1)$$

단일 노드에서 사용하는 해쉬 조인 알고리즘에는 GHJ (Grace Hash Join), SHJ(Simple Hash Join), 그리고 HHJ(Hybrid Hash Join) 등이 있다[9, 18]. GHJ는 가장 기본적인 알고리즘으로 분할 단계를 마친 버킷들은 디스크에 저장되고, 조인 단계에서는 디스크에 저장된 버킷들을 차례대로 읽어서 조인 연산을 실행한다. SHJ는 분할 단계에서 메모리 크기만큼 해쉬 테이블을 작성하여 조인 연산을 실행하며, 오버플로우가 발생한 데이타들은 디스크에 저장한다. 현재 실행 중인 조인 연산이 완료된 후 디스크에 저장된 데이타들에 대한 조인 연산을 반복적으로 실행한다. HHJ는 GHJ와 SHJ를 혼합한 알고리즘으로서, GHJ의 디스크 액세스 수를 줄이기 위하여 첫 번째 버킷(R_1)은 디스크에 저장하지 않고 메모리에서 해쉬 테이블을 바로 만든다. 그러므로 HHJ는 R_1에 대한 디스크 액세스 오버헤드를 줄일 수 있다.

2.2 부하분산 기법

해쉬 조인 알고리즘에서 버킷에는 동일한 해싱 값을 가지는 튜플만 저장되므로 버킷 단위의 병렬 조인 처리가 가능하지만, 버킷에 저장되는 튜플의 수가 균일하지 않은 경우(Data Skew)에는 특정 노드의 작업 지연으로 전체적인 질의 처리 시간이 길어질 수 있다. Data Skew를 해결하기 위한 부하분산 방법은 정적 부하분산 기법과 동적 부하분산 기법의 두 가지가 있다.

정적 부하분산 기법은 주로 DPS에서 사용되는 방법으로 조인 단계 실행 전에 노드간의 부하 균형을 맞추며 조인 단계에서는 부하분산 작업을 하지 않는다[4, 10, 11, 21]. [4]와 [11]의 경우, 분할 단계가 완료되면 각 노드에서 생성된 버킷 정보를 이용하여 버킷의 양이 많은 노드의 작업을 부족한 노드로 재분배한 후 조인 단계를 진행한다. [10]과 [11]은 분할 단계 전에 분석 단계를 두어 입력 릴레이션을 공평하게 분할할 수 있는 크기를 결정한다. 즉, 분석 단계에서 해쉬 함수를 이용하여 릴레이션을 분할한 결과를 히스토그램으로 만들고, 이를 이용하여 각 버킷들의 분할 크기를 결정한 후 분할 단계를 수행한다. 정적 부하분산 기법은 분할 단계에서 처리 노드가 결정되면 조인 단계 실행 중에는 더 이상의 작업의 분배가 없다. 따라서 결과 튜플들의 발생 빈도 차이로 인해 발생하는 또 다른 Data Skew(Join Product Skew : JPS)에 영향을 받을 수 있다. [10]의 경우, JPS를 해결하기 위하여 결과 튜플들을 또 다른 분할 테이블을 이용하여 여러 노드에 재분배한다. 그러나, 빈도수가 높은 애트리뷰트는 여러 노드에 저장되기 때문에 분할 테이블의 크기가 커질 수 있고 그에 따라 검색 속도가 늦어질 수 있다.

동적 부하분산 기법은 조인 단계에서 특정 노드의 작업이 많으면 다른 노드로 작업을 분산하여 각 노드들간의 작업 균형을 맞추는 방법이다[3, 8, 15, 19]. DPS에서는 노드들이 데이타베이스를 공유하지 않으므로 제한된 형태의 동적 부하분산 기법만 지원한다[3, 15, 19]. [3]은 각 데이타 프래그먼트의 액세스 빈도수를 측정하여 특정 임계치를 넘으면 해당 데이타를 재배치하는 방법으로 데이타가 여러 노드에 중복될 경우 효과적이다. [19]는 가상 공유 메모리를 이용하여 조인 단계 실행 중에 작업이 끝난 노드에서 아직 진행중인 노드의 작업을 분산하여 실행한다. [15]는 현재 노드들의 상태를 분석하여 동적으로 처리 노드와 처리 노드 수를 결정하지만, 조인 단계를 수행하는 동안에는 처리 노드가 고정되기 때문에 정적 부하분산 기법에 가깝다.

DSS의 경우 모든 노드가 데이타베이스를 공유하므로 동적 부하분산 기법을 적용하는 것이 훨씬 용이하다. DSS에서 동적 부하분산 기법을 적용한 유일한 연구로 Lu와 Tan의 연구[8]를 들 수 있다. Lu와 Tan의 연구에서 각 노드의 조인 처리 상태는 작업 테이블에 등록되며, 작업 테이블은 공유 메모리에 저장되어 모든 노드들이 액세스할 수 있다고 가정한다. 각 노드는 페이지 단위로 버킷 조인 연산을 실행한 후 작업 테이블에 해당 페이지의 식별자를 등록한다. 이때 다음으로 처리할 페이지의 식별자를 작업 테이블에서 액세스하여 해당 페이지의 조인 연산을 실행한다. 처리할 페이지가 존재하지 않는 경우, 작업 테이블에 등록된 정보를 바탕으로 작업이 많이 남아 있는 노드를 선택하고, 작업 테이블 내용을 변경한 후 해당 노드로부터 작업을 가져와서 실행한다.

Lu와 Tan의 연구의 단점으로는 공유 메모리에 대한 빈번한 액세스를 들 수 있다. 즉, 현재까지 상용화된 대부분의 DSS의 경우 공유 메모리가 하드웨어 차원에서 지원되지는 않는다. 따라서 공유 메모리를 구현하는 유일한 방법은 특정 노드가 작업 테이블에 대한 관리자 역할을 담당하며, 공유 메모리에 대한 다른 노드들의 액세스를 관리자와의 메시지 패싱으로 구현하는 것이다. 이럴 경우, 공유 메모리에 대한 페이지 단위의 빈번한 액세스는 과도한 메시지 패싱 오버헤드를 초래하며, 그 결과 동적 부하분산의 효과가 상쇄될 수 있다. 뿐만 아니라, Lu와 Tan의 연구에서는 GHJ 해쉬 조인 알고리즘에 대해서만 병렬화 방법을 제안했으며, 다른 해쉬 조인 알고리즘들에 대해서는 병렬화 방법 및 성능에 대한 연구가 이루어지지 않았다.

3. DSS에서 병렬 해쉬 조인 알고리즘

본 절에서는 노드들간의 공유 메모리가 존재하지 않는 일반적인 경우 작업 관리자와의 메시지 패싱 오버헤드를 줄일 수 있는 동적 부하분산 기법을 제안한다. 그리고, 제안한 동적 부하분산 기법을 이용하여 기존 해쉬 조인 알고리즘들의 병렬화 방안을 제안한다.

3.1 자료 구조 및 가정

본 논문에서는 동적 부하분산 정보를 "작업 관리자"

라는 별도의 노드가 관리한다고 가정하며, 질의를 처리하는 일반 노드는 작업 관리자와 메시지 패싱을 이용하여 통신한다고 가정한다. 작업 관리자가 관리하는 정보는 "버킷 정보"와 "프래그먼트 정보"로 나누어지며, 이러한 정보들은 "작업 테이블"에 저장된다고 가정한다.

버킷 정보는 [버킷 식별자(B_I), 버킷 크기(B_S), 처리되지 않고 남은 버킷 크기(B_U), 페이지 식별자 리스트(P_B)] 등으로 구성된다. B_S는 각 버킷의 크기를 페이지 단위로 나타낸 것이다. B_U는 노드에게 할당되지 않는 버킷 부분을 페이지 단위로 나타낸 것이며, 이 부분은 작업 관리자에 의해 각 노드에게 할당된다. P_B는 해당 버킷에 저장되는 페이지들의 식별자 리스트를 나타낸다. 각 노드는 분할 단계 후에 한번의 메시지 패싱으로 버킷 정보[B_I, B_S, B_U, P_B]를 작업 관리자에게 전송한다.

프래그먼트 정보는 [프래그먼트 식별자(F_I), 프래그먼트 크기(F_S), 작업의 할당 여부(F_A), 페이지 식별자 리스트(P_F)] 등으로 구성되는데, 하나의 버킷을 여러 노드로 할당하기 위해 프래그먼트 단위로 분할할 때 기록되는 정보이다. F_S는 각 프래그먼트의 크기를 페이지 단위로 나타낸 것이며, F_A는 할당된 해당 프래그먼트의 노드 할당 유무를 나타내는 플래그이다. P_F는 해당 프래그먼트에 포함되는 페이지들의 식별자 리스트를 나타낸다.

3.2 동적 부하분산 기법

동적 부하분산 기법은 해쉬 조인 알고리즘의 분할 단계와 조인 단계에 모두 적용된다. 먼저 분할 단계에서 각 노드는 자신에게 할당된 릴레이션의 부분들을 디스크에서 스캔하여 분할한다. 이때 다음절에서 설명할 해쉬 조인 알고리즘에 따라 분할되는 버킷 수 및 분할되는 버킷들의 처리 방법에 차이가 발생한다. 분할 단계가 끝나면 메시지 패싱을 통하여 생성된 버킷 정보[B_I, B_S, B_U, P_B]를 작업 관리자에게 전송한다. 작업 관리자는 작업 테이블에 모든 버킷 정보를 기록한다.

조인 단계에서는 B_U를 각 노드의 메모리 크기에 알맞게 분할한다. 본 논문에서는 노드의 메모리 크기를 동일한 것으로 가정하여 B_U를 동일한 크기의 프래그먼트로 분할한다. 만약 메모리 크기가 다르다면, 각 노드의 메모리 크기를 고려하여 서로 다른 크기의 프래그먼트들로 분할해야 한다.

동적 부하분산을 위한 작업 관리자의 처리 과정 및 조인 처리 노드들의 조인 단계에서 작업 과정이 (그림 1)과 (그림 2)에 각각 나타난다.

1. 각 노드로부터 버킷 정보[B_I, B_S, B_U, P_B]를 전송 받아 작업 테이블을 만든다.
2. B_U를 메모리 크기에 따라 프래그먼트 단위로 분할한다. 이때 프래그먼트 수(N)와 각 프래그먼트의 크기($F_S[i]$)는 다음과 같이 계산한다. 단, |M|은 메모리의 페이지 수이다.

$$N = \lceil B_U/|M| \rceil \quad (2)$$

$$F_S[i] = \begin{cases} \lceil B_u/N \rceil & i<N \\ B_s - \sum_{k=1}^{N-1} F_s[k] & i=N \end{cases} \quad (3)$$

이후 작업 테이블에 프래그먼트 정보[F_I, F_S, F_A, P_F]를 기록한다. F_A 플래그는 해당 프래그먼트가 아직 처리되지 않았으므로 '0'으로 설정한다.
3. 노드로부터 작업 요청 메시지가 전송되면, 할당되지 않은 프래그먼트를 작업 요청 노드에게 할당하고 [B_I, F_I, F_S, P_B, P_F] 정보를 포함한 메시지를 전송한다. 즉, 조인을 위한 내부 릴레이션(R)의 프래그먼트 정보 및 외부 릴레이션(S)의 버킷 정보를 전송한다. 이후 해당 프래그먼트의 F_A 플래그를 '1'로 설정한다.

(그림 1) 동적 부하분산을 위한 작업 관리자의 처리 과정

1. 작업 관리자에게 작업 요청 메시지를 보낸다.
2. 작업 관리자로부터 처리해야할 작업 메시지 [B_I, F_I, F_S, P_B, P_F]를 전송 받는다. 이때 더 이상 처리해야 할 작업이 없음을 알리는 메시지를 받으면 종료한다.
3. 조인 작업을 수행하기 위하여 릴레이션 R에서 P_F에 속하는 모든 페이지들을 디스크에서 액세스하여 해쉬 테이블을 만든다. 그리고, 릴레이션 S에서 P_B에 속하는 페이지들을 한 페이지씩 읽어서 R의 해쉬 테이블을 검색한다. 이때 R과 S의 조인 애트리뷰트(JA_R, JA_S) 값이 일치하는 튜플들이 조인 결과로 산출된다.
4. 단계 1로 진행한다.

(그림 2) 동적 부하분산을 위한 조인 처리 노드들의 조인 단계

다음의 [예 1]은 동적 부하분산 기법을 적용한 병렬 조인 알고리즘의 처리 과정을 나타낸다.

[예 1] 크기가 8 Kbyte인 페이지를 512개 저장할 수 있는 메모리(4Mbyte)를 가진 3개의 노드 N_1, N_2, N_3와 3개의 공유 디스크 D_1, D_2, D_3로 구성된 DSS를 가정하자. 이때, 릴레이션 R과 S의 크기는 각 32 Mbyte라고 가정한다.

단계 1 : 노드 N_1, N_2, N_3는 릴레이션 R에서 자신에게

할당된 부분들을 동시에 스캔하면서 해쉬 함수를 이용하여 분할한다. 또한 릴레이션 S도 같은 해쉬 함수를 이용하여 분할한다. 분할 단계가 끝난 후 각 노드는 릴레이션 R, S의 버킷 정보[B_I, B_S, B_U, P_B]를 작업 관리자에게 전송한다

단계 2 : 작업 관리자는 단계 1에서 각 처리 노드로부터 전송 받은 버킷 정보를 바탕으로 릴레이션 R의 버킷을 프래그먼트 단위로 분할하여 각 버킷에 해당하는 프래그먼트 정보를 기록한다. 예를 들면, 버킷 B의 B_U 값이 1200일 경우, 식 (2)와 식 (3)에 의해 프래그먼트 수 N은 3으로 결정되며, 각 프래그먼트의 크기(F_S)는 (그림 3)과 같이 결정된다. 각각의 프래그먼트들은 아직 할당되지 않았으므로 F_A의 값은 '0'이다.

F_I	F_S	F_A	P_F
1	400	0	400개의 페이지 리스트
2	400	0	400개의 페이지 리스트
3	400	0	400개의 페이지 리스트

(그림 3) 버킷 B의 프래그먼트 정보

[예 1] 끝

본 논문에서 제안한 동적 부하분산 기법과 Lu와 Tan의 연구[8]와의 차이점은 다음과 같다. 첫째, 조인 처리 노드와 작업 관리자간의 통신이 프래그먼트 단위로 발생한다는 것이다. 즉, 조인 처리 노드가 프래그먼트를 할당받은 후 프래그먼트에 속한 페이지들을 처리하는 과정에서는 메시지 전송이 발생하지 않는다. 따라서 작업 관리자와의 메시지 전송 횟수가 급격히 줄어든다. 물론 관련 페이지 식별자들을 포함하는 프래그먼트 할당 메시지의 크기는 Lu와 Tan의 연구에서 사용되는 페이지 할당 메시지의 크기보다 크다. 그러나, DSS가 구현되는 초고속 네트워크의 경우 네트워크를 통한 메시지 전송 시간보다 메시지 송·수신을 위한 CPU 처리 시간이 메시지 전송 시간의 대부분을 차지하므로, 메시지 전송 횟수를 줄임으로써 보다 큰 성능상의 이득이 발생할 수 있다.

또 다른 차이점은 Lu와 Tan의 연구의 경우 페이지 단위로 부하분산이 이루어지므로, 각 노드의 부하를 거의 동일하게 조정할 수 있다. 이와는 달리, 제안된 부하분산 기법에서는 특정 노드가 마지막 프래그먼트를 처리할 경우 작업이 없는 다른 노드에게 부하를 분산시킬 수 없으므로, 노드들간의 부하 차이가 다소 존재할 수 있다. 그러나, 프래그먼트는 메모리에 모두 저장될 수 있도록 분할되었으므로 프래그먼트에 대한 조인 연산은 상대적으로 짧은 시간에 처리될 수 있고, 그 결과 질의 응답 시간에 큰 영향을 미치지는 않는다.

3.3 병렬 해쉬 조인 알고리즘

본 논문에서는 기존의 해쉬 조인 알고리즘(GHJ, SHJ, HHJ)들을 동적 부하분산을 고려하여 DSS 환경으로 확장하였다. 본 절에서는 확장된 병렬 해쉬 조인 알고리즘들을 구체적으로 설명한다. 각 알고리즘의 정당성 증명 및 정성적 평가는 [22]를 참조하기 바란다.

3.3.1 Parallel Grace Hash Join(PGHJ)

PGHJ의 분할 단계는 다른 해쉬 조인 알고리즘들과 비교할 때 가장 단순하다. 즉, 각 노드는 자신에게 할당된 릴레이션 부분들에 대해 해쉬 함수를 적용하여 버킷을 생성하고, 버킷 오버플로우가 발생하면 디스크에 기록한다. 분할 단계가 완료되면 메모리에 남아있는 버킷 내용들을 디스크에 기록한 후, 작업 관리자에게 생성된 모든 버킷에 대해 [B_I, B_S, B_U, P_B]를 전송한다. 분할 단계에서는 버킷에 대한 조인 연산을 실행하지 않으므로 B_U는 B_S와 동일하다. 각각의 조인 처리 노드에서 실행되는 PGHJ 알고리즘의 구체적인 내용을 (그림 4)에 정리한다.

PGHJ에서 버킷 수(|B|)는 분할 단계를 위하여 메모리의 페이지 수(|M|)보다는 작아야 하며, 병렬성을 증가하기 위하여 노드 수보다는 커야 한다[5]. |B|가 클 경우 각 버킷의 크기(B_S)는 상대적으로 작아지는데, B_S가 |M|보다 작아지면 프래그먼트의 크기도 |M|보다 작아지고, 그 결과 조인 단계에서 메모리를 효율적으로 사용하지 못할 수 있다. 반대로 |B|가 작아져서 B_S가 커질 경우, 프래그먼트 단위로 분할하여 조인을 수행하므로 릴레이션 R에 대해서는 별도의 오버헤드가 발생하지는 않지만, 각 프래그먼트에 대해 릴레이션 S를 B_S만큼 액세스하여야 하므로 디스크 액세스 오버헤드가 증가한다. 본 논문에서는 |B|의 값을 "릴레이션의 페이지 수 / |M|"으로 설정하였다.

[예 1]의 경우, |B|는 4096/512 = 8로 설정되며 각 노드는 분할 단계에서 릴레이션 R과 S를 B_1~B_8의 8개 버킷으로 분할하여 디스크에 저장한다. 만약 R의 버킷 B_1의 B_U 값이 1200이면 작업 관리자는 (그림 3)과 같은 프래그먼트 정보를 생성하며, 각각의 프래그먼트는

PGHJ(Parallel Grace Hash Join)

분할 단계
1. 조인 애트리뷰트 JA_R을 해싱하여 R을 |B|개의 버킷으로 분할 (해쉬 함수 = h_1)
2. 분할된 |B|개의 버킷들을 디스크에 저장
3. 조인 애트리뷰트 JA_S를 동일한 해쉬 함수 h_1을 이용하여 |B|개의 버킷으로 분할
4. 분할된 |B|개의 버킷들을 디스크에 저장
5. R과 S의 버킷들에 대해 [B_I, B_S, B_U, P_B] 메시지를 작업 관리자에게 전송

조인 단계
1. 다음에 처리할 프래그먼트 정보를 작업 관리자에게 요청
2. while (작업 관리자가 [B_I, F_I ,F_S, P_B, P_F] 메시지 전송)
3.   for each (P_F에 속하는 R의 페이지 p)
4.   p를 디스크에서 판독
5.   p에 저장된 튜플 r을 해쉬 함수 h_2로 해싱하여 해쉬 테이블에 등록
6.   endfor
7.   for each (P_B에 속하는 S의 페이지 p)
8.   p를 디스크에서 판독
9.   p에 저장된 각각의 튜플 s를 h_2로 해싱하여 해쉬 테이블 검사
10.   동일한 해쉬 값을 갖는 r과 s에 대해 $r.JA_R = s.JA_S$일 경우, (r, s) 출력
11.   endfor
12.   다음에 처리할 프래그먼트 정보를 작업 관리자에게 요청
13. endwhile

(그림 4) PGHJ의 알고리즘

3개의 노드에서 S의 버킷 B_1과 동시에 조인될 수 있다.

(그림 4)의 경우 분할 단계와 조인 단계에서 상이한 해쉬 함수 h_1과 h_2를 이용하여 해싱 연산을 실행한다. 해쉬 함수 h_1은 릴레이션을 버킷 단위로 분할하기 위해 사용되며, h_2는 릴레이션 R의 프래그먼트와 S의 버킷들간에 동일한 조인 애트리뷰트 값을 갖는 튜플들을 빨리 검색하기 위하여 사용된다. h_1에 따라 버킷들의 크기가 달라질 수 있으므로 정적 부하 분산 기법의 경우 버킷의 크기가 유사하게 설정될 수 있도록 h_1을 결정해야 한다. 그러나, 본 논문에서 제안한 동적 부하 분산 기법의 경우, 크기가 큰 버킷은 여러 개의 작은 프래그먼트로 분할하여 동시에 조인 연산을 실행하므로 h_1에 따른 성능 차이는 상대적으로 크지 않다. 이런 관점에서 본 논문에서는 h_1을 버킷 수(|B|)에 대한 "나머지" 함수로 가정하였으며, h_2도 해쉬 테이블 크기에 대한 "나머지" 함수로 가정하였다.

3.3.2 Parallel Simple Hash Join(PSHJ)

SHJ를 DSS에 적용한 PSHJ의 경우, 버킷 수는 조인 연산에 참여하는 노드 수와 동일한 값으로 설정된다. 그리고, 각 노드는 분할 단계에서 릴레이션 R을 버킷 수만큼 분할하여 자신에게 할당된 버킷을 제외한 나머지 버킷들은 해당 노드로 전송한다. 이 때 버킷의 전송 단위는 페이지이다. 즉, 각 노드에서 해싱을 이용하여 버킷을 생성할 때, 특정 버킷의 크기가 한 페이지를 초과하게 되면 그 페이지를 버킷의 해당 노드로 전송한다.

각 노드는 버킷을 분할하는 과정 중에 다른 노드로부터 전송된 R의 페이지를 이용하여 해쉬 테이블을 생성한다. 이후 릴레이션 S에 해당하는 페이지가 전송되면 R의 해쉬 테이블을 검색하여 조인 결과를 산출한다. R의 해쉬 테이블을 생성할 때 메모리 오버플로우가 발생할 수 있고, 오버플로우 영역들은 디스크에 저장된다. 분할 단계가 완료한 후, 오버플로우가 발생한 버킷들에 대해 디스크에 저장된 부분(B_U)을 포함한 버킷 정보를 작업 관리자에게 전송하고, 이 부분들은 (그림 1)과 (그림 2)의 과정을 이용하여 처리된다.

PSHJ 알고리즘에 대한 자세한 내용이 (그림 5)에 나타난다. PSHJ의 조인 단계는 (그림 4)의 조인 단계와 동일하므로 (그림 5)에는 나타내지 않도록 한다.

PSHJ(Parallel Simple Hash Join)

분할 단계 - 분할
1. for each (처리되지 않은 R의 튜플 r)
2. $i = h_1(r.JA_R)$을 계산한 후, r을 버킷 R_i에 저장
3. R_i 크기가 페이지 크기와 동일할 경우, R_i를 노드 N_i로 전송
4. endfor
5. for each (처리되지 않은 S의 튜플 s)
6. $i = h_1(s.JA_S)$을 계산한 후, s를 버킷 S_i에 저장
7. S_i 크기가 페이지 크기를 초과할 경우, S_i를 노드 N_i로 전송
8. endfor

분할 단계 - 조인
1. while (모든 R_i 페이지를 수신)
2.   메모리 오버플로우가 발생할 경우, R_i 페이지를 디스크에 저장
3.   그렇지 않은 경우, R_i에 저장된 튜플 r을 해쉬 테이블에 등록 (해쉬 함수 = h_2)
4. endwhile
5. while (모든 S_i 페이지를 수신)
6. S_i에 저장된 튜플 s를 h_2로 해싱하여 해쉬 테이블에서 검사
7.   동일한 해쉬 값을 갖는 r과 s에 대해 $r.JA_R = s.JA_S$일 경우, (r, s) 출력
8. endwhile
9. R_i에 대해 [B_I, B_S, B_U, P_B] 메시지를 작업 관리자에게 전송
9.1 B_U: 디스크에 저장된 R_i 페이지 수
9.2 P_B: 디스크에 저장된 R_i 페이지들의 식별자 리스트
10. S_i에 대해 [B_I, B_S, B_U, P_B] 메시지를 작업 관리자에게 전송

(그림 5) PSHJ의 분할 단계 알고리즘

[예 1]에서 노드 수가 3이므로 버킷 수(|B|)는 3이다. 따라서, 노드 N_2는 노드 N_1에 버킷 R_1에 해당되는 페이지를 전송하고, 노드 N_3에 버킷 R_3에 해당되는 페이지를 전송한다. 만약 R_1의 크기가 1200일 경우, 메모리에 저장될 수 있는 508개의 페이지는 (그림 5)의 "분할 단계 - 조인" 과정에 의해 N_1의 분할 단계에서 버킷 S_1과 조인된다(분할을 위한 3개의 페이지와 조인을 위한 하나의 S_1 페이지를 가정할 경우). 디스크에 저장된 나머지 682개의 R_1 페이지에 대한 정보는 버킷 메시지에 의해 작업 관리자에게 통보되며, (그림 4)의 프래그먼트 조인 단계에서 처리된다. 이때 682개의 페이지는 2개의 프래그먼트로 나누어져 서로 다른 노드에서 동시에 S_1과 조인될 수 있다.

3.3.3 Parallel Hybrid Hash Join(PHHJ)

PSHJ와 마찬가지로 PHHJ도 분할 단계 중에 조인 연산을 일부 실행한다. PHHJ에서 버킷 수는 PGHJ와 동일하게 "릴레이션의 페이지 수/|M|"으로 설정된다. 단일 노드의 HHJ의 경우, 분할 단계에서 첫 번째 버킷에 대해서만 해쉬 테이블을 만들지만, PHHJ는 메모리 이용 효율을 증대시키기 위하여 처리 노드 수만큼의 버킷들에 대한 해쉬 테이블을 만든다. 즉, 각 노드는 릴레이션을 |B|개의 버킷($B_1 \sim B_{|B|}$)으로 분할하는데, 노드 수(N)만큼의 버킷($B_1 \sim B_N$)은 해당 노드로 전송하고 나머지 버킷($B_{N+1} \sim B_{|B|}$)들은 디스크에 저장한다.

각 노드는 전송된 릴레이션 R의 버킷에 대한 해쉬 테이블을 생성한 후, 릴레이션 S의 버킷이 전송되면 조인 연산을 실행한다. 해쉬 테이블을 생성할 때 오버플로우가 발생하면 PSHJ와 같이 오버플로우 영역들은 공유 디스크에 저장한다. 분할단계가 완료된 후, 각 노드는 오버플로우가 발생한 버킷 및 특정 노드에 할당되지 않은 버킷($B_{N+1} \sim B_{|B|}$)들에 대한 버킷 정보를 작업 관리자에게 전송한다. 이때, 특정 노드에 할당되지 않은 버킷들의 B_U는 B_S와 동일하다.

PHHJ 알고리즘에 대한 자세한 내용이 (그림 6)에 나타난다. PHHJ의 조인 단계는 (그림 4)의 조인 단계와 동일하며, 분할 단계에서의 조인 단계는 (그림 5)와 동일하므로 (그림 6)에는 나타내지 않도록 한다.

[예 1]의 경우, N은 3이고 |B|는 PGHJ에서와 같이 8이다. 따라서, 버킷 R_1, R_2, R_3는 해당 처리 노드로 전송되어 (그림 5)의 "분할 단계 - 조인" 단계에 의해 처리된다. 나머지 버킷들($R_4 \sim R_8$)은 디스크에 저장되고, 분할 단계가 끝난 후 버킷 정보가 작업 관리자로 전송된다. 결국, (그림 4)의 프래그먼트 조인 단계에서는 R_1, R_2, R_3의 "분할 단계 - 조인"에서 메모리 오버플로우가 발생하여 디스크에 저장된 부분과, "분할 단계 - 분할"에서 디스크에 저장된 전체 버킷들($R_4 \sim R_8$)을 분할하여 생성된 프래그먼트들에 대해 조인 연산을 실행한다.

PHHJ(Parallel Hybrid Hash Join)

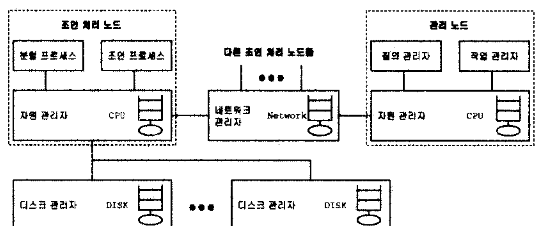
분할 단계 - 분할
1. for each (처리되지 않은 R의 튜플 r)
2. i = $h_1(r.JA_R)$을 계산한 후, r을 버킷 R_i에 저장
3. 만약 $i \le N$이며 R_i 크기가 페이지 크기와 동일할 경우,
   R_i를 노드 N_i로 전송
4. 만약 $i > N$이며 R_i 크기가 페이지 크기와 동일할 경우,
   R_i를 디스크에 저장
5. endfor
6. for each (처리되지 않은 S의 튜플 s)
7. i = $h_1(s.JA_S)$을 계산한 후, s를 버킷 S_i에 저장
8. 만약 $i \le N$이며 S_i 크기가 페이지 크기와 동일할 경우,
   S_i를 노드 N_i로 전송
9. 만약 $i > N$이며 S_i 크기가 페이지 크기와 동일할 경우,
   S_i를 디스크에 저장
10. endfor
11. $i > N$인 R_i에 대해 [B_i, B_S, B_U, P_B] 메시지를 작업 관리자에게 전송 (B_S = B_U)
12. $i > N$인 S_i에 대해 [B_i, B_S, B_U, P_B] 메시지를 작업 관리자에게 전송

(그림 6) PHHJ의 분할 단계 알고리즘

4. 성능 평가 모형

본 논문에서 제안한 해쉬 조인 알고리즘들의 성능 평가를 위해 개발한 DSS 모형이 (그림 7)에 나타난다. 모의 실험은 미국의 MCC에서 개발한 CSIM 언어[17]를 이용하여 수행되었다.

(그림 7)에서 조인 처리 노드들은 네트워크를 통해 연결되어 있고 디스크를 공유한다. 각각의 조인 처리



(그림 7) 성능 평가 모형

노드는 분할 단계를 담당하는 분할 프로세스와 조인 연산을 처리하기 위한 조인 프로세스로 구성된다. 분할 프로세스 및 조인 프로세스의 동작 과정은 해쉬 조인 알고리즘에 따라 구현된다. 자원 관리자는 자신에게 할당된 CPU 작업을 모델링하며, 네트워크를 통한 다른 조인 처리 노드들과의 버킷 전송 및 관리 노드와의 통신 과정을 수행한다. 관리 노드는 동적 부하분산 기능을 담당하는 작업 관리자와 조인 처리 노드에게 릴레이션 분할을 할당하는 질의 관리자로 구성된다. 관리 노드의 기능을 특정 조인 처리 노드에서 수행하는 방법도 가능하지만, 본 논문에서는 조인 알고리즘의 성능을 명확히 나타내기 위하여 별개의 노드에서 관리 기능을 수행한다고 가정한다.

본 논문에서 사용한 입력 매개 변수를 <표 1>에 요약한다. 각 매개 변수의 구체적인 값은 [8, 12]에서 참조하였으며, 본 논문에서 수행한 대부분의 실험에서 이 값들을 사용하였다. <표 1>의 값과 다른 값을 사용하는 실험의 경우, 실험 결과를 소개하기 전에 이를 먼저 설명하도록 한다.

〈표 1〉 입력 매개 변수

시스템 구성 변수		
CPUSpeed	노드의 CPU의 처리 속도	10 MIPS
MemSize	노드의 메모리 버퍼 크기	4 Mbyte
NetBandwidth	네트워크의 데이타 전송 속도	10 Mbps
NumNode	조인 처리 노드의 수	4~40
RelSize	릴레이션의 크기 (튜플 수)	1000 K
DistinctTuple	구별되는 튜플 수	10 K
NumDisk	디스크 수	16
DiskTime	디스크 액세스 시간	0.01~0.03 초
PageSize	페이지의 크기	8 Kbyte
TuplesPerPage	페이지에 포함되는 튜플 수	50
오버헤드 변수		
FixedMsgInst	메시지 처리 위한 고정 명령 수	20,000
PerByteMsgInst	메시지 페이지 당 추가 명령 수	10,000
PerIOInst	디스크 I/O를 위한 명령 수	5,000
ReadTuple	튜플을 읽기 위한 명령 수	500
WriteTuple	튜플을 버퍼에 쓰기 위한 명령 수	500
ProbeHashTable	해쉬 테이블을 검색하는 명령 수	200
InsertHashTable	해쉬 테이블 삽입하는 명령 수	100
HashTuple	튜플을 분할하는 명령 수	100

조인 처리 노드의 수 및 메모리 크기를 변화시키면서 실험을 수행하였으며, 모든 노드의 CPU 성능은 동일한 것으로 가정하였다. 조인 처리 노드의 수를 변화할 때 메모리 크기는 4 Mbyte로 고정하였다. 공유 디스크의 수는 16개로 고정하였으며, 디스크 액세스 시간은 0.01초에서 0.03초까지의 일양 분포를 따른다. 디스크는 FIFO 큐를 이용하여 I/O 요청이 발생된 순서대로 처리한다. 또한, 분할 단계에서 I/O 성능을 증가시키기 위하여 각 릴레이션은 라운드 로빈(round-robin) 방식으로 디스크에 분산되어 저장된다[2].

네트워크 관리자는 10 Mbps의 대역폭을 갖는 FIFO 서버로 구현되었다. 네트워크를 통해 메시지를 전송하는 과정을 표현하기 위해 메시지마다 FixedMsgInst 만큼의 고정된 명령 수와 메시지 크기가 한 페이지를 넘을 경우 페이지 당 PerByteMsgInst 만큼의 추가적인 명령 수를 실행한다.

릴레이션 R과 S의 튜플 수는 1000 K인데, 페이지 크기가 8 Kbyte이고 페이지에 들어가는 튜플 수가 50이므로 R과 S의 페이지 수는 각각 20480이다. 각 페이지에 들어가는 조인 애트리뷰트의 값은 Zipf 분포[6]를 따른다. 릴레이션 R의 조인 애트리뷰트는 D라는 도메인 영역을 가진다. D의 범위는 $1 \le i \le$ DistinctTuple이고, i의 값을 가지는 튜플 수는 식 (4)에 의해 계산된다. 이때 $\|R\|$은 릴레이션 R의 튜플 수를 의미한다. θ의 값이 0인 경우는 균일(Uniform) 분포에 해당되고, θ의 값이 1인 경우는 Data Skew 정도가 높은 비 균일(Non-Uniform) 분포에 해당한다.

$$\| D_i \| = \frac{\| R \|}{i^{\theta} \sum_{j=1} \frac{1}{j^{\theta}}} \quad (4)$$

병렬 질의 처리의 목적이 응답 시간을 줄이는 것이므로 본 논문에서는 조인 연산의 응답 시간을 주요 성능 평가 지수로 한다. 응답 시간에는 큐에서의 대기 시간 및 디스크 I/O 시간, 통신 지연 시간도 모두 포함된다. 신뢰성 있는 모의 실험 결과를 얻기 위해 배치 평균 기법(batch mean method)을 사용하였다. 즉, 본 논문에서 나타난 실험 결과들은 20개의 다른 seed를 이용하여 산출된 결과들의 평균값이다.

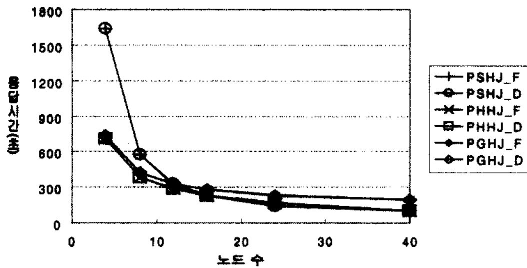
5. 실험 결과 분석

앞 절에서 설명한 성능 평가 모형을 이용하여 수행한 모의 실험 결과들을 분석한다. 동적 부하분산의 효과를 분석하기 위해 각 해쉬 조인 알고리즘들을 동적 부하분산을 지원하는 경우와 지원하지 않는 경우의 두

가지 형태로 구현하였다. 프래그먼트 단위의 동적 부하분산을 지원하는 경우 해쉬 조인 알고리즘에 따라 PGHJ_D와 PSHJ_D, 그리고 PHHJ_D로 표현하며, 버킷 단위로 부하를 분산하는 해쉬 조인 알고리즘들은 PGHJ_F와 PSHJ_F, 그리고 PHHJ_F로 표현한다. 다양한 시스템 구성 및 데이타베이스 부하 상태에서 알고리즘들의 성능을 분석하기 위하여 조인 처리 노드 수와 각 노드의 메모리 크기, 그리고 Data Skew 정도를 변화시키면서 실험을 수행하였다.

5.1 실험 1 : 균일 분포 - 노드 수의 변화

본 절에서는 Data Skew 정도가 동등한 균일 분포에서 조인 처리 노드 수의 변화에 따른 해쉬 조인 알고리즘의 성능을 비교한다. 균일 분포는 Zipf 분포에서 $\theta=0$인 경우이다. 각각의 노드는 512개 페이지를 처리할 수 있는 고정된 크기의 메모리(4 Mbyte)를 갖는다. (그림 8)은 노드 수를 4에서 40까지 변화하면서 실험한 해쉬 조인 알고리즘들의 성능 평가 결과를 나타낸다.



(그림 8) 균일 분포에서 노드 수의 변화

균일 분포에서 프래그먼트 단위의 동적 부하분산을 지원하는 경우와 그렇지 않은 경우에 대해 해쉬 조인 알고리즘의 성능이 거의 동일하였다. 그 이유는 균일 분포이므로 분할 단계에서 생성된 버킷의 크기가 큰 차이가 없고, 그 결과 버킷 단위의 부하분산에서 특정 노드에게 부하가 편중되는 경우가 발생하지 않기 때문이다.

PSHJ는 노드 수가 증가할수록 성능이 급속도로 향상되어, 노드 수가 4인 경우에 비해 노드 수가 40인 경우 16배 성능이 향상되었다. 그 이유는 다음과 같다. PSHJ는 분할 단계에서 분할된 버킷을 디스크에 저장하지 않고 처리 노드로 전송하여 해쉬 테이블로 바로 만들기 때문에, 조인 단계의 실행 횟수는 ⌈ |B| / |M| ⌉에 비례한다[18]. 여기서 |B|는 릴레이션 R의 버킷 당 페이지 수이고 |M|은 노드 메모리의 페이지 수이다. 노드 수가 증가할수록 버킷 수가 늘어나고 각 버킷의 크기는 줄어들며, 그 결과 조인 단계의 실행 횟수 및 소요 시간이 줄어든다.

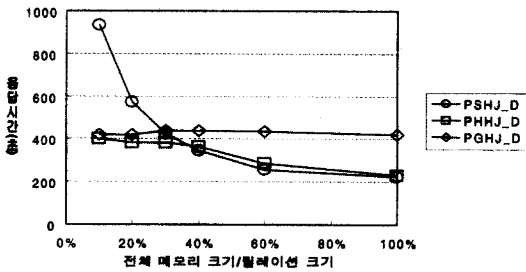
PSHJ와는 달리 PGHJ는 노드 수의 증가에 따른 성능 향상 정도가 상대적으로 적게 나타났다. 그 이유는 PGHJ에서 생성되는 버킷 크기는 각 노드의 메모리 크기에만 영향을 받으므로, 버킷 당 조인 처리 시간은 노드 수에 관계없이 동일하기 때문이다. 그러나, 노드 수가 늘어날수록 조인 단계의 병렬성이 증가하므로 전체적인 응답 시간은 줄어든다. 노드 수가 매우 클 경우 PGHJ는 분할 단계의 결과를 디스크에 기록하는 오버헤드로 인해 PSHJ보다 성능이 조금 떨어졌다.

PHHJ는 노드 수가 적을 경우 PGHJ와 비슷한 성능을 나타내며, 노드 수가 많을 경우 PSHJ와 성능이 유사하였다. 즉, 버킷 크기가 메모리 크기에 따라 결정되므로 노드 수가 아주 적을 경우에도 버킷 당 조인 처리 시간이 급격히 증가하지 않으며, 노드 수가 많을 경우에는 분할 단계에서 대부분의 버킷을 디스크에 저장하지 않고 바로 조인할 수 있다.

5.2 실험 2 : 균일 분포 - 메모리 크기 변화

본 절에서는 조인 처리 노드의 메모리 크기를 변화하면서 조인 알고리즘들의 성능을 분석한다. 노드 수는 8개로 고정되고 전체 메모리 크기는 릴레이션 크기의 10%에서 100%까지 변화한다. PHHJ와 PGHJ는 메모리 크기에 따라 버킷 수가 달라지는데, 메모리 크기가 10%인 경우는 버킷 수가 80이고 100%인 경우는 8이다. 실험 1의 결과, 균일 분포에서는 부하분산 기법에 따른 해쉬 조인 알고리즘들의 성능 차이가 발생하지 않으므로 본 절에서는 동적 부하분산 기법의 결과만을 설명한다. 균일 분포에서 메모리 크기에 따른 성능 평가 결과가 (그림 9)에 나타난다.

PSHJ_D는 메모리 크기 변화에 영향을 많이 받았다. 즉, 메모리 크기가 10%인 경우 나머지 알고리즘들에 비해 성능이 2배 이상 저하되었으며, 100%인 경우는 PGHJ_D보다 성능이 2배정도 좋고 PHHJ_D와는 성능이 유사하였다. 그 이유는 PSHJ_D의 경우 메모리 크기가 작으면 버킷 오버플로우가 빈번하게 발생하고, 그 결과 프래그먼트 단위의 조인 단계 실행 횟수가 늘어



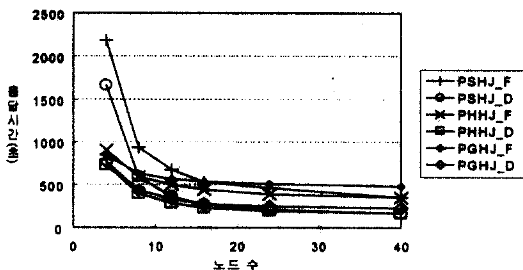
(그림 9) 균일 분포에서 메모리 크기 변화

난다. 이때 각 조인 단계에서 특정 프래그먼트와 조인되는 릴레이션 S의 버킷 크기는 PGHJ_D나 PHHJ_D에 비해 크다. 따라서 메모리 크기가 작을 경우 PSHJ_D의 프래그먼트에 대한 조인 처리 시간이 매우 길어진다. 메모리 크기가 증가함에 따라 PSHJ_D의 조인 단계 실행 횟수가 줄어들어 성능이 급격히 향상된다.

PHHJ_D는 전 구간에 걸쳐서 PGHJ_D에 비해 성능이 좋다. 그 이유는 메모리 크기가 커질수록 분할 단계에서 디스크에 저장되지 않고 바로 조인될 수 있는 릴레이션 부분들이 증가하기 때문이다. 그 결과 메모리 크기가 10%인 경우는 약간의 성능 차이를 보이는 반면에, 메모리 크기가 100%인 경우는 2배정도 성능이 향상되었다.

5.3 실험 3 : 비 균일 분포 - 노드 수 변화

본 절에서는 Data Skew 정도가 높은 비 균일 분포에서 조인 처리 노드 수를 변화하면서 해쉬 조인 알고리즘들의 성능을 비교한다. 비 균일 분포는 Zipf 분포에서 $\theta=1$인 경우이다. (그림 10)에 실험 결과가 나타난다.

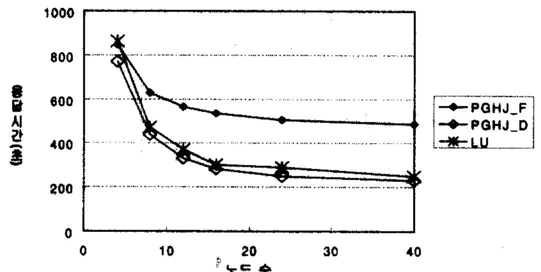


(그림 10) 비 균일 분포에서 노드 수의 변화

모든 알고리즘들에 대해 프래그먼트 단위의 동적 부하분산 기법이 버킷 단위의 부하분산 기법보다 20%에서 100% 정도 성능이 우수한 것으로 나타났다. 그 이유는 비 균일 분포이므로 특정 버킷의 크기가 다른 버킷들에 비해 매우 커질 수 있고, 버킷 단위로 조인 연산을 할당할 경우 큰 버킷을 할당받은 조인 처리 노드에 의해 조인 응답 시간이 길어질 수 있기 때문이다. 이와는 달리, 프래그먼트 단위로 조인 연산을 분할할 경우 큰 버킷에 대한 조인 연산을 여러 노드에서 동시에 처리할 수 있으므로 노드들 간의 응답 시간 편차가 줄어든다.

흥미로운 사실은 노드 수가 증가할수록 부하분산 기법에 따른 성능 차이의 정도가 커진다는 것이다. 그 이유는 명확하다. 즉, 프래그먼트 단위의 동적 부하분산 기법의 경우, 노드 수가 증가할수록 보다 많은 노드에서 큰 버킷의 프래그먼트들을 동시에 처리할 수 있다. 그러나, 버킷 단위의 부하분산 기법에서는 버킷을 처리하는 노드가 고정되므로, 노드 수가 증가하더라도 조인 연산을 완료한 노드들을 다른 노드에게 할당된 버킷 조인 연산에 활용할 수 없다.

Lu와 Tan의 연구[8]와 본 논문에서 제안한 동적 부하분산 기법들간의 성능 차이를 분석하기 위하여 [8]에서 제안한 병렬 GHJ 알고리즘을 구현하여 PGHJ_D와 PGHJ_F의 성능과 비교하였다. (그림 11)에 그 결과가 나타난다. "LU"는 [8]에서 제안한 병렬 GHJ 알고리즘을 의미한다.



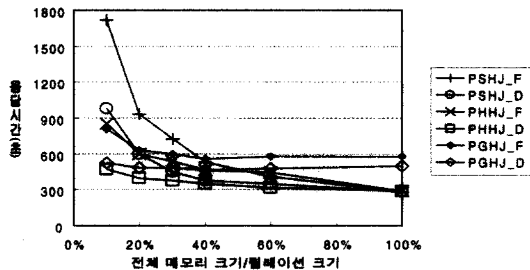
(그림 11) 비 균일 분포에서 노드 수 변화에 따른 동적 부하분산 기법의 성능

LU 알고리즘의 경우 동적 부하분산을 위하여 작업 관리자와 조인 처리 노드들간에 페이지 단위의 메시지 전송이 발생한다. 따라서 메시지 전송 오버헤드가 커지며, 그 결과 PGHJ_D에 비해 성능이 약 10% 정도 저하되었다. PGHJ_D의 경우 프래그먼트 단위로 메시

지 전송이 발생하므로 메시지 전송 오버헤드는 줄일 수 있지만, 조인 단계의 마지막 부분에서 프래그먼트를 할당받지 못한 노드가 존재할 수 있으므로 노드들 간의 부하 불균형이 다소 발생할 수 있다. 그러나, 프래그먼트는 메모리에 모두 저장될 수 있는 크기로 설정되었으므로 프래그먼트 단위의 조인 연산은 상대적으로 짧은 시간에 처리될 수 있고, 그 결과 빈번한 메시지 전송으로 인한 성능 저하 정도보다 성능에 미치는 정도가 낮게 나타났다.

5.4 실험 4 : 비 균일 분포 - 메모리 크기 변화

본 절에서는 비 균일 분포에서 조인 처리 노드의 메모리 크기를 변화하면서 해쉬 조인 알고리즘들의 성능을 비교한다. 실험 결과가 (그림 12)에 나타난다.



(그림 12) 비 균일 분포에서 메모리 크기의 변화

메모리 크기가 작을 경우, 프래그먼트 단위의 동적 부하분산 기법이 버킷 단위의 부하분산 기법보다 최대 100% 정도 성능이 우수한 것으로 나타났다. 그 이유는 메모리 크기가 작으면 버킷 오버플로우가 빈번하게 발생하고, 버킷 단위의 부하분산을 수행할 경우 크기가 매우 큰 버킷을 할당받은 노드에 의해 전체 질의 응답 시간이 지연되기 때문이다. 프래그먼트 단위의 부하분산 정책에서는 버킷 오버플로우가 발생한 부분들을 다른 노드에서 처리할 수 있으므로, 노드들간의 조인 처리 연산의 편차가 줄어든다. 메모리 크기가 늘어날수록 부하분산에 따른 성능 차이가 줄어드는데, 그 이유는 버킷 오버플로우의 크기 및 발생 빈도수가 줄어들기 때문이다.

균일 분포의 결과와 마찬가지로 PGHJ는 메모리 크기가 늘어나도 성능이 크게 향상되지는 않았다. PSHJ와 PHHJ는 분할 단계에서 조인 연산을 처리하므로, 메모리 크기가 늘어날 경우 디스크 액세스 빈도수가 줄어든다. 그러나, PGHJ는 분할 단계의 결과가 반드시 디스크에 기록되어야 하므로 메모리 크기에 관계없이 고정적으로 디스크 액세스가 발생한다.

Lu와 Tan의 연구[8]에서 제안한 알고리즘과 본 논문에서 제안한 방법에 대해 메모리 크기를 변화시키면서 실험을 수행하였다. 논문의 길이 제한에 의해 실험 결과는 포함시키지 않았으나, 전체적인 성능 차이는 (그림 11)과 유사한 형태로 나타났다. 즉, PGHJ_D 알고리즘이 LU 알고리즘에 비해 성능이 약간 향상되었다. 자세한 실험 결과는 [22]를 참조하기 바란다.

6. 결 론

대부분 기존 병렬 조인 알고리즘들은 DPS를 가정하여 제안되었다. 그러나, DPS에 비해 DSS는 모든 노드들이 데이타를 공유하므로 부하분산이 용이하고, 그 결과 병렬 조인 처리에 훨씬 적합하다는 장점을 갖는다. 이러한 관점에서 본 논문은 DSS에서 프래그먼트 단위의 동적 부하분산 기법을 제안하고, 이를 이용하여 해쉬 조인 알고리즘들을 DSS 환경으로 확장하였다. 제안된 부하분산 기법 및 해쉬 조인 알고리즘들의 성능을 평가하기 위하여 다양한 시스템 구성 및 데이타베이스 부하 환경에서 모의 실험을 수행하였다.

주요 실험 결과는 다음과 같이 요약할 수 있다. 첫째, Data Skew 정도가 높은 비 균일 분포의 경우, 프래그먼트 단위의 동적 부하분산 기법은 버킷 단위의 정적 부하분산 기법에 비해 최대 100% 정도 성능이 좋은 것으로 나타났다. 둘째, DSS를 구성하는 노드 수가 증가할수록 동적 부하분산 기법의 성능 향상 정도가 커졌다. 이러한 결과는 병렬 시스템의 규모가 커지고 있는 추세에 비추어 볼 때 바람직한 결과라고 판단된다. 마지막으로, 모든 실험에서 PHHJ의 성능이 가장 우수한 것으로 나타났다. PSHJ는 노드 수가 많을 경우나 메모리가 큰 경우 PHHJ와 성능이 유사한 것으로 나타났으며, PGHJ는 실험 환경 변화에 따른 성능 변화의 정도가 상대적으로 완만했다.

참 고 문 헌

[1] M. Abdelguerfi and K. Wong, *Parallel Database Techniques*, IEEE Comp. Society Press, 1998.

[2] D. DeWitt and J. Gray, "Parallel Database Sys-

ems : The Future of High Performance Database Systems," *Comm. ACM*, Vol.35, No.6, pp.85-98, 1992.

[3] A. Heal, A. Yuan and H. Rewni, "Dynamic Data Reallocation for Skew Management in Shared-Nothing Parallel Databases," *Distributed and Parallel Databases*, Vol.5, No.3, pp.271-288, 1997.

[4] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash : A New, Robust, Parallel Hash Join Method for Data Skew in Super Database Computer (SDC)," *Proc. 16th Int. Conf. VLDB*, pp.210-221, 1990.

[5] M. Kitsuregawa, et al., "The Effect of Bucket Size Turning in the Dynamic Hybrid GRACE Hash Join Method," *Proc. 15th Int. Conf. VLDB*, pp.107-120, 1989.

[6] D.E. Knuth, *The Art of Programming, Vol.3 : Sorting and Searching*, Addison Wesley, 1973.

[7] H. Lu, B. Ooi and K. Tan, *Query Processing in Parallel Database Systems*, IEEE Comp. Society Press, 1994.

[8] H. Lu and K. Tan, "Dynamic and Load-Balanced Task-Oriented Database Query Processing in Parallel Systems," *Proc. 3rd Int. Conf. Extending Database Tech*, pp.357-372, 1992.

[9] P. Mishra and M. Eich, "Join Processing in Relational Databases," *ACM Comp. Surveys*, Vol. 24, No.1, pp.63-113, 1992.

[10] E. Omiecinski, "Performance Analysis of a Load-Balancing Relational Hash Join Algorithm for a Shared-Memory Multiprocessor" *Proc. 17th Int. Conf. on VLDB.*, pp.375-385, 1991.

[11] U. Park, H. Choi and T. Kim, "Uniform Partition of Relation Using Histogram Equalization Framework : An Efficient Parallel Hash-Based Join," *Information Processing Letters*, 55(5), pp. 283-289, 1995.

[12] V. Poosala and Y. Ioannidis, "Estimation of Query-Result Distribution and its Application in Parallel-Join Load Balancing," *Proc. 22nd Int. Conf. VLDB*, pp.447-459, 1996.

[13] E. Rahm, "Parallel Query Processing in Shared Disk Database Systems," *Proc. 5th Int. Conf. High Performance Transactions Syst.*, 1993.

[14] E. Rahm, "Analysis of Parallel Scan Processing in Shared Disk Database Systems," *Proc. EURO-PAR Conf.*, 1995.

[15] E. Rahm and R. Marek, "Dynamic Multi-Resource Load Balancing in Parallel Database Systems," *Proc. 21th Int. Conf. VLDB*, pp.395-406, 1995.

[16] D. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared Nothing Multiprocessor Environment," *Proc. ACM SIGMOD*, pp.110-121, 1989.

[17] H. Schwetman, *CSIM Users Guide for use with CSIM Revision 16*, MCC, 1992.

[18] L. Shapiro. "Join Processing in Database Systems with Large Main Memories," *ACM Trans. on Database Syst.*, Vol.11, No.3, pp.239-264, 1986.

[19] A. Shatdal and J. Naughton, "Using Shared Virtual Memory for Parallel Join Processing," Proc. ACM SIGMOD, pp.119-128, 1993.

[20] C. Walton, A. Dale, and R. Jenevin, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins," *Proc. 17. Int. Conf. VLDB*. pp.536-548, 1993.

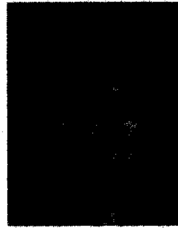
[21] J. Wolf, P. Yu, J. Turek and D. Dias, "A Parallel Hash Join Algorithm for Managing Data Skew," *IEEE Trans. on Parallel and Distributed Syst.*, 4(12), pp.1355-1371, 1993.

[22] 문애경, 데이타베이스 공유 시스템에서 병렬 조인 처리 기법, 영남대학교 박사학위논문, 1999.

문 애 경

e-mail : akmoon@cse.yu.ac.kr

1992년 영남대학교 전산공학과 학사
1997년 영남대학교 전산공학과 석사
1997년~현재 : 영남대학교 컴퓨터공학과 박사과정



조 행 래

e-mail : hrcho@ynucc.yu.ac.kr

1988년 서울대학교 컴퓨터공학과 학사
1990년 한국과학기술원 전산학과 석사
1995년 한국과학기술원 전산학과 박사
1995년~현재 : 영남대학교 컴퓨터정보통신공학부 조교수
관심분야 : 분산/병렬 데이타베이스, 트랜잭션 처리, DBMS 개발 등