멀티코어 아키텍처에서 프리페칭 기법을 통한 해시조인 구현1)

신재현*, 김재명**, 이상원*
*성균관대학교 임베디드소프트웨어학과
**성균관대학교 정보통신공학부
e-mail:{jhyshin, jam02, swlee}@skku.edu

Implementing Hash Join through Prefetching on Mulit-core Architecture

Jae-Hyun Shin*, Jae-Myung Kim**, Sang-Won Lee*
*Dept of Embeded Software, Sung-Kyun-Kwan University
**Dept of ICE, Sung-Kyun-Kwan University

요 약

해시 조인 알고리즘 성능 개선에 관한 연구는 이미 많은 연구자에 의해 수행된 바 있다. 새로운 알고리즘을 추가하는 연구에서부터 컴퓨팅 환경에 맞는 최적화 솔루션을 제시하는 연구에 이르기까지 해시 조인의 성능을 향상시키는 연구는 다양하게 찾아 볼 수 있었다. 이 논문에서는 2004년 ICDA에서 발표한 [1]의 연구를 최신의 컴퓨팅 환경에서도 똑같이 작동하는지 확인해 보고자 한다.

1. 서론

해시 조인은 등가조인(equi-join)을 실행하는 대표적인 조인 기법의 한 종류로, 많은 상용 데이터베이스 시스템에서 사용되고 있다.

해시 조인은 조인을 실행하고자 하는 두 개의 테이블의 크기 차이가 클 때 효과적으로 사용 된다. 해시 조인의 원리는 다음과 같다. 우선, 크기가 작은 테이블을 빌드 테이블로 설정하고, 크기가 큰 테이블을 프로브 테이블로 설정한다. 빌드 테이블의 조인키를 이용하여 해시 테이블을 구성하고, 프로브 테이블을 조인키로 해시 테이블에서 매칭되는 값을 찾아서 조인한다.

이 논문은 [1]에서 제안된 해시조인에 소프트웨어 프리페칭 기법을 적용한 연구의 아이디어를 구현하였다. 또한 [1]을 참고하여 다중 쓰레드 컴퓨터 환경에서 실험을 수행한 [2]를 참고하였다. [1]의 연구에서는 해시조인 알고리즘에 소프트웨어 프리페칭 기법을 사용하여 2.0~2.9배 성능 개선을 이뤄냈다. 그러나 그 실험은 소프트웨어로 시뮬레이션된 프로세서에서 수행되어 실제 컴퓨팅 환경에서 완벽하게 적용되는지 알 수 없었고, 이를 보완하여 [2]에서 실제 컴퓨팅 환경에서 [1]과 동일한 실험을 수행하여 1.4 ~ 1.55배 성능개선을 확인하였다.

[1], [2]의 연구가 수행된 이후 컴퓨팅 환경이 매우 많이 변경되었고, 다중 코어와 늘어난 캐시 환경을 갖춘 컴퓨팅 환경에서도 [1], [2]의 연구가 유효한지 검증하고자 실험을 수행하였다. 우리 실험에서는 동일한 실험에서 0.96 ~ 1배로 성능향상을 볼 수 없었다. 아직 검증해야 하는 추가 실험이 남아 있지만, 성능 개선을 볼 수 없었던 이유를 통해 또 다른 해시 조인 성능 개선 요소를 찾을 수 있을 것으로 기대된다.

1) 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 육성지원사업의 연구결과로 수행되었음 (NIPA-2012-(H0301-12-3001))

2. 관련 연구

GRACE 해시 조인 알고리즘 일반적인 해시조인은 해시 테이블 빌드 단계와 빌드된 해시 테이블에서 매칭되는 키값을 찾는 프로브 단계로 이뤄진다. 빌드 테이블의 크기가 메인 메모리 크기보다 커지게 되면, 일반적인 해시 조인을 사용할 수 없게 된다.

이러한 해시 조인의 한계를 극복하고자 새로운 알고리즘이 제안 되었다. [3]에서 처음 소개된 GRACE 해시 조인 알고리즘(이하 GRACE)은 크게 두 개의 실행 단계로 이뤄진다. 첫 번째는 파티션 단계이며, 두 번째는 조인 단계이다. [1]에서 두 단계를 완전히 구분하기 위해 GRACE를 사용하였고, [1], [2]에서 모두 성능 개선을 비교하기 위한 기본 알고리즘으로 사용하였다. 본 연구에서도 마찬가지로 GRACE를 기본 알고리즘으로 사용한다.

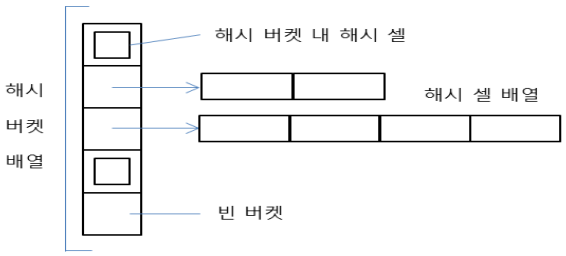
해시 조인의 CPU 캐시 스톨(stall) 현상 GRACE는 해시 연산의 랜덤성으로 인해 공간적 지역성이나 시간적 지역성을 기대할 수 없다. 이것은 각 튜플의 조인키값으로 해시 연산하게 되면, 결과인 버켓 인덱스가 버켓의 개수만큼 넓은 분포로 랜덤성을 가지게 된다. 따라서 해시 연산된 버켓 인덱스 주위의 버켓을 같이 메모리 참조하는 것은 아무런 의미가 없기 때문에 공간적 지역성을 기대할 수 없다. 마찬가지 이유로 동일한 버켓 인덱스가 연속해서

호출된 확률은 지극히 낮다. 그렇기 때문에 시간적 지역성 또한 기대할 수 없다. 이로 인해 CPU 캐시의 다중 구조의 이점을 취할 수 없다.

[1]에서 GRACE의 파티션 단계와 조인 단계 각각의 실행 시간을 분해한 결과, 각각 실행 시간의 82%와 73%가 데이터 캐시 실패로 인해 스톨된 시간이다. 이는 해시 조인 알고리즘이 등가조인에 좋은 성능을 보이고 있음에도 불구하고 많은 개선이 필요함을 단적으로 보여주는 예이다.

캐시 프리페칭 해시 조인 알고리즘에서 캐시 스톨 현상은 해시 테이블을 아주 작게(캐시 사이즈로) 하여 캐시에 모두 올릴 수 있게 하면 CPU 캐시 실패를 피할 수 있다. 그러나 대용량의 테이블의 캐시 사이즈로 파티션하는 것은 많은 I/O 손실이 발생하여 성능 개선의 효과를 기대할 수 없다. 그래서 [1]에서는 메인 메모리 사이즈에 적합한 해시 테이블을 이용하여 성능 개선을 할 수 있는 방법을 제안하였다. 이것을 캐시 실패를 없애는 것 대신 캐시 실패에 의한 지연을 캐시 프리페칭을 이용하여 지연 시간동안 CPU가 다른 일을 할 수 있도록 하는 방법을 제안하였다. 그 방법은 SSE(Streaming SIMD Extensions)와 같은 명령어 세트에서 제공하는 사전인출(Prefetch) 명령어를 사용하여 캐시에 미리 데이터를 옮겨 놓는 방법이다.

그룹 프리페칭 기법(Group Prefetching) 해시 테이



(그림 1) 메모리 속 해시 테이블 구조

블의 구조는 (그림1)과 같다. 그림과 같이 포인터 연결 구조를 가지고 있다. 단일 튜플의 프로브 단계에서 포인터 추적 경로는 조인 키값으로 해시 코드를 계산하여 해당 해시 버킷을 찾은 후 해당 해시 셀을 방문한다. 이후 해시 셀에 연결된 셀 배열을 추적하면서 조인할 수 있는 튜플을 찾는다. 단일 튜플은 하나의 조인 튜플을 찾기 위한 프로세스가 단계별로 의존성을 가지고 있다. 그러나 해시 조인의 특성상 하나의 튜플을 처리하는 과정과 또 다른 튜플과 처리 과정은 의존성이 전혀 존재하지 않는다. 그룹 프리페치는 이러한 개별 튜플간 데이터 의존성이 없다는 점을 이용하여 만든 알고리즘이다. 이 그룹 프리페치는 조인하기 위한 튜플 간에 분리된 메모리 접근을 이용한다. 우선 몇 개(설정한 그룹 수만큼)의 튜플을 동시에 메모리에 할당하여 그룹을 만든다. 그룹 내의 각 튜플은 반복문을 실행하면서 해시 조인을 위한 첫 번째 연산을 동시에 실행하고, 동시에 다음 단계에서 해당 튜플이 접근하여 처

```
프로브 파티션 내 그룹 단위 반복문
{
    그룹 내 튜플 단위 반복문
    {
        해시 버켓 숫자 계산;
        계산된 버켓 헤더 프리페치;
    }
    그룹 내 튜플 단위 반복문
    {
        해시 버켓 헤더 접근;
        해시 버켓에 연결된 해시 셀 프리페치;
    }
    그룹 내 튜플 단위 반복문
    {
        해시 셀 배열 접근;
        비교하는 빌드 튜플 프리페치;
    }
    그룹 내 튜플 단위 반복문
    {
        비교하는 빌드 튜플 접근;
        키값 비교와 출력 튜플 생성;
    }
}
```

(그림 3) 프로브 단계의 그룹 프리페칭 의사코드

리해야 하는 메모리의 데이터에 대해 프리페치 명령어를 실행한다. 그룹 내에 튜플들의 동일한 연산을 실행하고 프리페치 하는 시간은 한 개의 튜플에서 해시 연산으로 인한 메모리 접근 지연시간을 감춰주는 역할을 한다. 이러한 그룹 연산을 각 단계별로 동일한 방법으로 실행하고, 계속해서 이후 튜플들에 대한 그룹을 만들어 반복해 나간다. (그림 2)는 프로브 단계에서 그룹 프리페치를 실행하는 의사코드이다.

3. 실험 설정

<표 1> AMD 컴퓨터 구성현황

CPU	AMD Opteron 6128 (8 × 2.0GHz)
L1 캐시	8 × 64 KB 명령어 캐시 8 × 64 KB 데이터 캐시
L2 캐시	8 × 512 KB (private)
L3 캐시	2 × 6 MB (shared)
메인 메모리	5 × 4 GB (20GB)
OS	Linux 2.6.32 (64bit)
컴파일러	gcc 4.4.6 -O3
측정도구	CPU 클럭 카운터 : perf 2.6.32 경과 시간 : gettimeofday()

GRACE와 그룹 프리페치를 실행한 실험 환경은 <표 1>과 같다. 구현한 알고리즘은 GRACE과 이 GRACE의 소스코드에서 프로브 단계의 알고리즘만 그룹 프리페치를 적용하여 구현하였다. 이 두 가지 프로그램을 이용하여 기본 GRACE과 그룹 프리페치의 성능 차이를 비교하였다.

실험에 사용한 빌드 테이블과 프로브 테이블 정보는 <표 2>와 <표 3>과 같다. 프로브 테이블의 키값은 빌드 테이블의 키값의 범위에서 랜덤하게 부여하였고, 100% 빌드 테이블에 매칭되도록 하였다. 파티션 직전 원 테이블을 디스크에서 읽는 것을 제외하고 디스크에서 읽기와 쓰기는 없다.

<표 3> 실험 테이블 구성

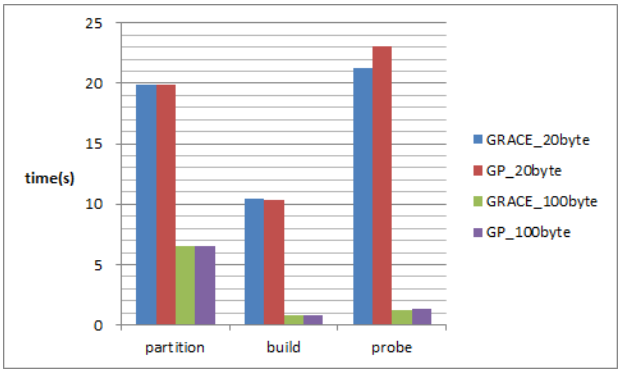
튜플 사이즈	테이블명	크기	튜플수
100B	빌드테이블	200MB	1,991,200
	프로브테이블	400MB	3,982,396
20B	빌드테이블	200MB	9,576,720
	프로브테이블	400MB	19,153,494

<표 2> 실험 테이블 스키마

테이블명	컬럼	데이터형	크기	속성
빌드 테이블	A	double	8B	primary key
	B	char	20~100B	value
프로브 테이블	A	double	8B	key
	B	char	20~100B	value

4. 실험 결과

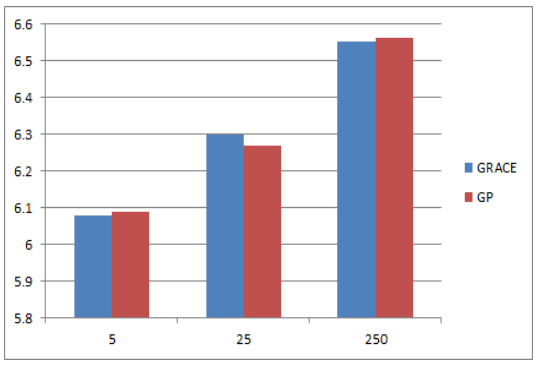
튜플 사이즈 변경에 따른 경과시간 변화 튜플 사이즈를 각각 20B와 100B로 설정하여 200MB, 400MB의 테이블을 만들어 실험한 결과는 (그림 3)와 같다(범례의 GP는 그룹 프리페치, 이하 그림의 GP는 동일한 의미임). 파티션 단계와 빌드 단계는 동일한 소스코드를 사용하여 GRACE와 그룹 프리페치 모두 동일한 결과를 볼 수 있다. 프로브 단계에서 GRACE 해시 조인과 그룹 프리페치를 적용한 알고리즘의 성능 비교는 [1]에서 보인 것처럼 그룹 프리페치를 적용한 알고리즘이 더 좋을 거라 예상했으나, 실제로 그룹 프리페치가 0.93 ~ 0.96배 느린 결과가 나왔다.



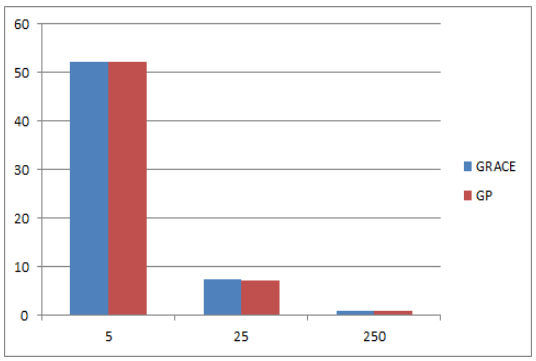
(그림 3) 튜플 크기별 단계별 경과시간 비교

레코드 수가 약 5배 정도 차이나지만, 빌드 단계와 프로브 단계에서는 100B 테이블을 실행한 프로그램들이 5배 이상 빠른 성능을 보였다. 빌드 단계에서는 디스크 읽기로 인해 약 3배 정도 성능차가 보인다.

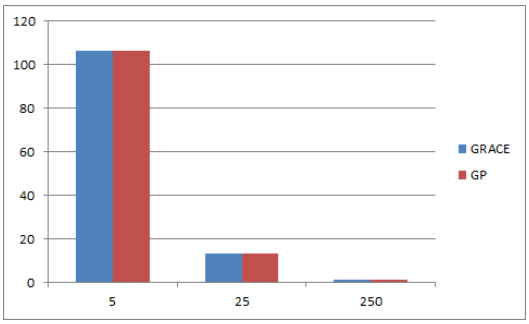
파티션 개수 변경에 따른 경과시간 변화 이 실험에서 사용한 테이블은 튜플 사이즈가 100B인 테이블을 사용하였다. 파티션 단계에서는 파티션의 개수가 늘수록 경과시간이 늘어나는 것을 (그림 4)에서 확인할 수 있다. 동일한 소스코드이지만, GRACE와 그룹 프리페치가 실행 시 컴퓨터 상태에 따른 차이가 0.01~0.03 초정도로 근소하게 차이를 보이고 있다.



(그림 4) 파티션 단계



(그림 5) 빌드 단계



(그림 6) 프로브 단계

빌드 단계에서도 마찬가지로 두 알고리즘 모두 동일한 소스코드이기 때문에 동일한 결과가 나온 것을 (그림 5)에서 확인할 수 있다. 파티션 수가 늘어남에 따라 경과시간이 약 650% 감소(파티션 개수 5와 250의 차이)하는 것을 확인할 수 있다.

프로브 단계는 [1]의 결과와 같이 GRACE와 그룹 프리페치의 차이를 볼 수 있기를 기대하고 있었으나, 두 알

고리즘 간의 차이를 거의 볼 수 없었다. (그림 6)에서 파티션이 늘어날수록 경과시간이 840%(파티션 개수 5와 250의 차이) 감소 줄어드는 것은 확인 할 수 있으나, 두 알고리즘 간의 시간차는 0.05~0.4초로 거의 근소한 결과를 볼 수 있다.

5. 결론 및 향후 연구

[1]에서는 소프트웨어 CPU 시뮬레이션을 이용하여 실험한 결과 GRACE보다 그룹 프리페치가 2.0~2.9배의 성능 향상을 보였다. [2]에서는 실제 펜티엄4(Prescott) CPU를 사용하여 실행했을 때 그룹 프리페치가 1.4 ~ 1.55배 성능이 좋은 것으로 결과를 보였다. 처음 이 실험을 계획했을 때에도 이런 성능 향상을 확인하기 위한 것이었으나, 우리의 실험 환경에서는 GRACE와 그룹 프리페치의 성능 차이를 거의 볼 수 없었다. 오히려 GRACE가 1%미만의 근소한 차이로 더 좋은 성능을 보이고 있다.

이러한 결과를 보이는 이유를 몇 가지 추정할 수 있는데, 우선은 컴파일러의 명령어 스케줄링 최적화가 과거 실험했을 때보다 많이 좋아졌다는 점을 들 수 있겠다. 명령어 스케줄링에서 CPU 지연이 예상되면, 바로 다음 명령어를 한참 뒤에 두는 최적화는 이미 일반화 되어 있다.

우리 실험의 결과는 추가 실험으로 다시 검증해 봐야 할 것이다. 또한 여러 가지 가설을 세워 검증해 봐야 할 것이다.

참고문헌

[1] S. Chen, A. Ailamki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In IEEE International Conference on Data Engineering, 2004.

[2] P. Garcia and H. F. Korth. Hash-join algorithms on modern multithreaded computer architectures. In ACM Int'l Conf. on Computing Frontiers, May 2006.

[3] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and its Architecture. New Generation Computing, 1(1):63□ 74, 1983.