ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 42, No. 3, pp. 379-389, 2015. 3
http://dx.doi.org/10.5626/JOK.2015.42.3.379

PRAM 기반의 조인 알고리즘 성능 비교 연구
(A Comparative Study of PRAM-based Join Algorithms)

최 용 성 † 온 병 원 †† 최 규 상 ††† 이 인 규 ††††
(Yongsung Choi) (Byung-Won On) (Gyu Sang Choi) (Ingyu Lee)

요 약 Phase Change Memory (PCM 또는 PRAM), Magneto Resistive RAM (MRAM)과 같은 차세대 비휘발성 메모리가 등장하면서, Dynamic Random-Access Memory (DRAM)을 PRAM으로 대체하는 연구가 활발히 진행되고 있다. 본 논문에서는 PRAM을 메인 메모리로 사용하는 시스템에서 지금까지 널리 사용되고 있는 기존의 조인 알고리즘(블록 네스티드 조인, 소트-머지 조인, 그레이스 해시 조인, 하이브리드 해시 조인)들을 사용했을 때 발생하는 내구성과 성능 문제를 비교, 분석한다. 본 연구의 실험 결과에 의하면 기존의 조인 알고리즘들을 PRAM에 맞게 재설계해야 하는 필요성이 제기되었다. 특히, 본 연구는 조인 알고리즘들을 PRAM에 적용했을 때 발생하는 이슈들을 과학적으로 규명한 첫 시도이다. 그리고 기존의 조인 알고리즘들을 PRAM에 적용했을 때 발생하는 내구성과 성능을 비교하기 위한 PRAM 기반의 시스템을 모델링하고 시뮬레이터를 구현한 것에 연구의 의의를 둘 수 있다.
키워드: PRAM, 조인, 내구성, 비휘발성 메모리

Abstract With the advent of non-volatile memories such as Phase Change Memory (PCM or PRAM) and Magneto Resistive RAM (MRAM), active studies have been carried out on how to replace Dynamic Random-Access Memory (DRAM) with PRAM. In this paper, we study both endurance and performance issues of existing join algorithms that are based on PRAM-based computer systems and have been widely used until now: Block Nested Loop Join, Sort-Merge Join, Grace Hash Join, and Hybrid Hash Join. Our experimental results show that the existing join algorithms need to be redesigned to improve both the endurance and performance of PRAMs. To the best of our knowledge, this is the first research to scientifically study the results of the four join algorithms running on PRAM-based systems. In this work, our main contribution is the modeling and implementation of a PRAM-based simulator for a comparative study of the existing join algorithms.
Keywords: PRAM, join, endurance, non-volatile memories

· 이 논문은 2013년도 정부(미래창조과학부)의 한국연구재단의 일반연구자지원사업(No. 2013012524)의 연구비 지원과 정부(교육부)의 재원으로 한국연구재단의 기초연구사업(No. 2013R1A1A2063304)의 연구비 지원으로 수행하였습니다.

† 학생회원 : 경기대학교 컴퓨터과학과
soonmewar@naver.com
†† 종신회원 : 군산대학교 통계컴퓨터과학과 교수
(Kunsan National Univ.)
on.byung.won@gmail.com
(Corresponding author임)
††† 종신회원 : 영남대학교 정보통신공학과 교수
castchoi@ynu.ac.kr
†††† 종신회원 : 차세대융합기술연구원 스마트그리드연구센터
inlee3941@gmail.com

논문접수 : 2014년 9월 4일
(Received 4 September 2014)
논문수정 : 2014년 12월 30일
(Revised 30 December 2014)
심사완료 : 2015년 1월 6일
(Accepted 6 January 2015)

정보과학회논문지 제42권 제3호(2015. 3)

1. 서 론

전원이 차단되더라도 데이터가 유실되지 않는 비휘발성 메모리가 등장하면서 컴퓨터의 메인 메모리로 사용되던 DRAM을 대체하려는 연구가 활발히 진행되고 있다. 표 1은 다양한 차세대 비휘발성 메모리들의 특성을 보여준다[1]. 현재 널리 사용되는 낸드 플래시 메모리는 DRAM보다는 오히려 하드디스크를 대체하는 저장 매체로 각광 받고 있다. 현재 여러 개의 낸드 플래시 메모리를 멀티채널로 연결하여 높은 성능을 보이는 Solid State Disk (SSD)가 보편화되고 있으며, 낸드 플래시 메모리는 비휘발성 메모리이면서 속도가 빠르며, 저전력, 작은 폼팩터 등의 장점을 가지고 있다. 최근에는 낸드 플래시 메모리로 구성된 64GB의 대용량 저장장치를 가지는 모바일 디바이스의 확산으로 인해 낸드 플래시 메모리의 사용이 크게 증가하고 있다. Ferroelectric RAM (FRAM)은 낸드 플래시 메모리 보다 훨씬 뛰어난 내구성과 성능을 보여 주지만, 집적도가 매우 낮고, 단가도 매우 높기 때문에, 낸드 플래시 메모리를 대체하기 보다는 DRAM을 대체하는 것을 목표로 하고 있다. 이에 반하여, MRAM은 PRAM과 매우 유사한 특성을 가지고 있으며, 낸드 플래시 메모리를 대체하는 것을 목표로 하고 있지만, 아직 상용화되기 위해서는 많은 연구가 필요한 실정이다. 이와 같이 다양한 비휘발성 메모리가 등장하고 있지만, 상용화되고 있거나 이에 근접한 메모리는 낸드 플래시 메모리와 PRAM뿐이다.

낸드 플래시 메모리는 페이지 단위로 읽고 쓰기가 가능하다. 그리고 페이지의 집합인 블록 단위로 삭제(erase)가 가능하다. 다시 말하면, 블록에 포함된 데이터를 업데이트하기 위해서는 블록 전체를 삭제하고 메모리의 다른 장소에 업데이트 내용을 씀으로써 업데이트를 수행한다. 이것은 주기적으로 삭제 연산을 발생시키는 시간 소모가 큰(time-consuming) 연산이다. 또한 읽기(read)와 쓰기(write) 간의 연산 시간이 불균형하다. 반면에 PRAM은 바이트 단위로 업데이트가 가능하다. 또한 낸드 플래시 메모리에 비해 읽기/쓰기 지연시간(read and write latency time)과 내구성(endurance) 등에서 높은 성능을 보이는 것으로 알려져 왔다[2]. 그러나 DRAM에 비해 낸드 플래시 메모리와 PRAM은 각 메모리 셀에 쓰기 횟수가 제한된다. 낸드 플래시 메모리의 내구성은 100,000번으로 한 메모리 셀에 그 이상 쓰기를 할 수 없다. 반면에 PRAM의 내구성은 낸드 플래시 메모리보다 약 10배 정도 높으며, 읽기와 쓰기 시간도 낸드 플래시 메모리에 비해 월등히 빠르다. 현 시점에서는 PRAM이 낸드 플래시 메모리를 대체할 가장 유력한 비휘발성 메모리이며, 현재까지 대부분의 연구들이 PRAM을 DRAM 대용으로, 또는 DRAM과 함께 사용하는 하이브리드 방식으로 구성되어 성능을 향상시키는 연구에 집중되어 왔다.

PRAM의 내구성을 향상시키기 위해 각 메모리 셀의 쓰기 횟수를 균등하게(wear leveling) 분배함으로써, 메모리의 수명을 향상시키는 것이 필요하다. 그러나 기존의 많은 응용 프로그램들은 내구성 문제가 없는 DRAM 기반으로 설계되었기 때문에, PRAM을 메인 메모리로 사용할 경우, PRAM의 수명을 떨어뜨리고 짧은 시간 내에 PRAM을 새로 교체해야하기 때문에 많은 비용이 들어가게 된다. 더욱이 PRAM과 낸드 플래시 메모리는 전혀 다른 특성을 가지고 있기 때문에, 낸드 플래시 메모리에 최적화된 알고리즘들을 PRAM에 사용할 경우 오히려 성능을 떨어뜨릴 수 있다. 따라서 PRAM을 메인 메모리로 사용하는 시스템에서 동작하는 컴퓨터 알고리즘은 쓰기 연산을 메모리 전체에 균등하게 분배하면서 데이터를 처리할 수 있도록 재설계되어야 한다.

본 논문에서는 다양한 조인 알고리즘들을 PRAM에서

표 1 메모리별 성능 비교

Table 1 Comparison of memory performance

	DRAM	NAND Flash	FRAM	MRAM	PRAM
Non-volatile	No	Yes	Yes	Yes	Yes
Storage unit	Capacitor	Floating gate	Capacitor	MTJ	Phase change layer
Type	Electric charge	Electric charge	Polarization	Resistance size	Resistance size
Retention of Information	0yr	10yr	10yr	10yr	10yr
Write latency	50ns	1~2μs	50~100ns	10~50ns	50ns
Read latency	50ns	20~110ns	50~100ns	10~50ns	50ns
Cell	Area (relative)	1	0.8	1.3	1
Number of write operations	10^{15}	10^{5}	10^{12} - 10^{15}	10^{12}	10^{6}
Commercialization density	1GB	1GB	256KB	N/A	128MB
Operating current	100mA	100mA	10mA	10mA	10mA
Logic integration	Hard	Hard	Easy	Easy	Easy
Applications	PC	Mobile	IC card	Anywhere	Anywhere

사용할 경우에 발생하는 내구성과 성능 이슈를 실험을 통해 자세히 비교, 분석한다. 특히, 조인(join) 알고리즘은 메모리의 용량보다 저장장치에 저장되어 있는 데이터의 크기가 매우 커서 데이터의 일부를 메모리에 가져와서 두 테이블(table) 간의 조인 연산을 수행한다. 이 경우에 메모리에서 처리된 중간 결과를 다시 저장장치에 저장하는 프로세스를 반복한다. 조인 알고리즘은 대용량 데이터를 처리하는데 가장 중요한 알고리즘이며 반복적으로 데이터를 메모리에 업데이트하는 특징을 가진다. 만일 내구성에 제한이 있는 PRAM을 사용할 경우에 균등 분배를 고려하지 않는다면, PRAM의 내구성은 현저히 악화되고, 급기야 PRAM의 수명이 짧아져서 새로운 PRAM으로 교체해야 되며, 이에 따른 비용이 증가하는 것은 자명하다. 본 논문에서는 PRAM 기반의 시스템을 모델링한 시뮬레이터(simulator)를 구현하고, Block Nested Loop Join, Sort-Merge Join, Grace Hash Join, Hybrid Hash Join 알고리즘을 실행시켜 얻어진 결과를 바탕으로 PRAM의 내구성과 알고리즘의 성능을 평가하고 분석을 수행한다.

본 논문의 구성은 다음과 같다. 2장에서는 비휘발성 메모리와 관련된 기존 연구에 대해 살펴보고, 3장에서는 다양한 조인 알고리즘들에 대해 논의한다. 4장에서는 실험 결과를 비교, 분석하고, 5장에서 결론을 맺는다.

2. 관련연구

2.1 낸드 플래시 메모리의 균등 분배 기법 연구

낸드 플래시 메모리의 수명을 높이기 위하여 다양한 균등 분배 기법들이 제안되었다. 가능하면, 삭제(erase) 연산이 발생하지 않도록 하는 가비지 컬렉션(garbage collection)이 적게 발생하도록 하는 기법들과 전체 낸드 플래시 메모리가 비슷하게 사용되기 위하여 효과적인 쓰기 횟수 관리 기법과 쓰기 횟수가 적은 블록을 효율적으로 찾을 수 있는 기법들이 제안되었다[3].

2.2 낸드 플래시 메모리 기반의 B+ 트리 연구

FlashDB[4]은 실제로 B+ 트리를 구현하기 위해, 노드의 저장 방식을 크게 두 가지 방식인 B+ 트리(디스크)와 B+ 트리(로그)로 사용하는 것을 제안하였다. B+ 트리(디스크)는 일반적인 파일 시스템과 같은 방식으로 읽기(read) 연산을 40% 정도 빠르게 할 수 있고 B+ 트리(로그)는 로그 스트럭처(log-structured) 파일 시스템과 같은 방식으로 쓰기(write) 연산을 80% 정도 빠르게 할 수 있다. 트리의 각 노드에 대해, B+ 트리(디스크) 또는 B+ 트리(로그) 인덱스를 사용할 지를 결정할 수 있다. 예를 들면, 루트 노드와 가까운 노드들은 일반적으로 인덱스 노드들로 리프 노드들에 액세스하기 위해 보통 읽기 연산을 주로 함으로써, 이러한 노드들에 대해 B+ 트리(디스크) 인덱스를 사용한다. 반면에 리프 노드에 대해서 B+ 트리(로그)를 사용하는 것을 제안하였다. 아울러, B+ 트리(디스크)와 B+ 트리(로그)의 선택이 워크로드의 특성에 따라서 실시간으로 선택되도록 하는 셀프 튜닝(self-tuning) 기법을 제안하였다.

μ-tree[5]는 기본적으로 B+ 트리 인덱스 구조를 변경하지 않으면서, 단순히 B+ 트리 상에 있는 노드들을 메모리 페이지에 저장되는 위치를 낸드 플래시 메모리에 적합하게 변경하는 기법을 제안하였다. 이 연구의 핵심 아이디어는 B+ 트리의 루트 노드부터 리프 노드 상의 패스(path)에 위치하며 업데이트되는 노드들을 하나의 낸드 플래시 메모리 페이지에 저장하도록 하여 쓰기 횟수를 최소화한 것이다. 다만, 이 방식의 단점은 소량의 임베디드 데이터베이스에는 적합하지만, B+ 트리의 크기가 매우 커지는 대용량의 데이터베이스에는 적합하지 않다는 것이다.

FD-tree[6]는 낸드 플래시 메모리의 특성을 활용하여 임의(random) 쓰기 연산을 줄이고, 순차(sequential) 쓰기 연산을 늘리는 것을 최대한 활용하는 방식을 제안하였다. 이 논문에서는 기존의 B+ 트리를 두 영역으로 구분한다. 첫 번째 영역은 루트 노드를 포함하여 루트 노드로부터 i 레벨까지를 기본의 B+ 트리를 이용하여 구성하며, 두 번째 영역은 기존의 B+ 트리와는 완전히 다른 방법으로 구성한다. 이 논문에서 두 번째 영역에서는 각각의 레벨에 있는 모든 노드들이 연속적으로 한 공간에 쓰이도록 하여, i 레벨에서는 중간에 i+1 레벨 노드의 특정한 위치를 포인팅 하도록 하여, 특정한 i 레벨이 업데이트될 경우에 i+1 레벨이 영향 받지 않도록 하는 것이 핵심 아이디어이다. 이 방법에서는 새로운 데이터의 추가는 루트 노드에서부터 시작한다. 그래서 새로운 데이터가 첫 번째 영역에 추가된다면, 첫 번째 영역에서만 업데이트가 발생한다. 즉 두 번째 영역에는 업데이트가 발생하지 않는다. 그 이유는 첫 번째 영역에서 두 번째 영역에 대한 위치 정보를 가지고 있음으로 이 정보들이 업데이트되지 않기 때문이다. 다만 이 방식은 최악의 경우에 전체 트리에서 업데이트가 발생할 수 있으며, 버퍼 사용으로 인하여 비정상적인 전원이 끊어졌을 때, 버퍼에 있는 데이터를 잃어버릴 수 있는 단점을 가지고 있다.

2.3 DRAM을 대체하기 위한 PRAM 연구

Lee et al[7]는 DRAM 대신에 PRAM을 사용하기 위해, PRAM 내에 여러 개의 버퍼를 두거나, PRAM의 수명을 증가시키기 위해 부분적인 쓰기 기법을 활용하는 것을 제안하였다. 이러한 기법들을 사용하면, 기존의 DRAM과 비교하여 1.2 배의 성능 지연이 있지만, PRAM의 수명이 5~6년 보장되는 것으로 알려졌다.

Queresh et al[2]는 PRAM의 성능을 높이기 위하여, 다양한 구조적 디자인 이슈들을 비교 및 분석하였으며, PRAM의 수명을 증가시키기 위하여, DRAM과 PRAM을 함께 사용하는 혼합 구조를 제안하였다. 이러한 기법들을 사용하여, PRAM의 수명이 9년으로 연장되었고, 3배의 성능 향상 되는 것을 알 수 있다.

Flip-N-Write[8]은 PRAM의 수명을 증가시키기 위해, 기존의 데이터와 새로운 데이터 간의 차이를 비트 단위로 비교하여, 변경된 데이터만을 업데이트하도록 하는 기법이다. 이와 같은 방법으로 기존의 쓰기 횟수와 비교하여 최대 63%의 쓰기 횟수가 감소하는 것으로 나타났다.

2.4 PRAM과 낸드 플래시 메모리를 함께 사용하는 하이브리드 연구

PFFS[9]은 PRAM과 낸드 플래시 메모리를 함께 사용하는 연구로, 낸드 플래시 메모리의 메타 데이터를 낸드 플래시 메모리를 대신하여 PRAM에 저장하는 방식을 제안하였다. 메타 데이터의 업데이트는 대부분 몇 바이트만의 업데이트를 필요로 하기 때문에, 낸드 플래시의 메타 데이터를 낸드 플래시 메모리 자체에 저장하면 메타 데이터를 업데이트하기 위하여 낸드 플래시 메모리의 한 페이지를 다 써야 하는 오버헤드가 크다. 하지만 워드 단위로 읽기와 쓰기가 가능한 PRAM에 메타 데이터를 저장하면 매우 효율적이다. 따라서 이 연구에서 제안하는 방식은 기존의 낸드 플래시 메모리를 사용하는 시스템에 비하여 25%의 성능 향상을 보여 주었다. 하지만, 이 연구에서는 일반 데이터에 비하여 메타 데이터가 훨씬 자주 업데이트가 발생하므로, PRAM에 낸드 플래시 메모리의 메타 데이터를 저장함으로써 PRAM의 수명에 대한 문제를 해결해야 하지만, 완벽하게 해결하지는 못했다.

Nath와 Kansal[10]은 PRAM과 낸드 플래시 메모리를 함께 사용한 점에서 위의 연구와 비슷하지만, 이 연구에서는 파일 시스템과 낸드 플래시 메모리의 메타 데이터를 모두 PRAM에 저장하는 점이 다르다. 이 연구는 대략 290%의 높은 성능 향상을 보여주었지만, PRAM의 내구성 문제를 고려하지 않은 한계를 가지고 있다.

2.5 요약

기존의 PRAM 연구는 DRAM을 대체했을 때 발생하는 여러 문제들을 주로 하드웨어 관점에서 다루었다. 본 논문에서는 소프트웨어 관점에서 기존의 조인 알고리즘을 PRAM에 적용했을 때 발생하는 내구성과 성능 이슈에 대한 심도 있는 연구를 다루고 있으며, 이것은 기존 연구에서 다루어지지 않은 부분이다.

3. PRAM 기반의 조인 알고리즘

조인 알고리즘은 대용량의 두 테이블[1]을 조인(join)하기 위해 사용된다. 특히 테이블의 크기가 방대해서 메모리에 모두 넣을 수 없기 때문에, 데이터의 일부를 메모리로 가져와서 처리하고, 다시 저장장치에 저장함으로써 조인 연산을 수행한다. 현재까지 다양한 종류의 조인 알고리즘들이 개발되었다. 현재 널리 알려진 알고리즘으로는 Block Nested Loop Join, Sort-Merge Join, Grace Hash Join, Hybrid Hash Join 등이 있다[11]. 본 논문에서는 위의 4개의 조인 알고리즘들을 PRAM 기반의 시스템에서 사용했을 때 발생하는 성능과 내구성 문제를 살펴보고 그 결과를 비교, 분석한다.

먼저 PRAM 기반의 시스템은 CPU, 메인 메모리, 자장장치로 구성된다. 메인 메모리는 휘발성 메모리인 DRAM 대신에 PRAM을 사용한다. 이를 통해 갑작스런 전원 차단에도 불구하고 메인 메모리에 있는 데이터는 유실되지 않고 계속 사용할 수 있기 때문에, 최근 각광을 받고 있는 인메모리 데이터베이스(in-memory database)에서 활용 가치가 매우 높다. 메인 메모리 사이즈는 32KB로 가정하고[2], 메모리 공간은 크게 R 테이블 영역(8KB), S 테이블 영역 (8KB), Input Buffer (4KB), Output Buffer (4KB), 기타 영역 (8KB)으로 나눈다. 편의상, R 테이블 영역, S 테이블 영역, 입력 버퍼(input buffer), 출력 버퍼(output buffer), 기타 메모리 영역을 M_R, M_S, M_I, M_O, M_H라고 표시한다. M_H 영역은 Sort-Merge Join에서는 데이터를 정렬하기 위해 사용되며, Grace Hash Join과 Hybrid Hash Join에서는 해시 테이블을 저장하기 위한 공간으로 사용된다.

R 테이블은 <Department_ID, Department_Name>와 S 테이블은 <Employee_ID, Department_ID>의 스키마를 가진다. S 테이블에서 Employee_ID는 기본키(primary key)이고, Department_ID는 외래키(foreign key)이다. 또한 R 테이블에서 Department_ID는 R 테이블의 기본키이고 Department_Name은 value라고 할 수 있다. 따라서 조인 연산은 Select R.Employee_ID, S.Department_ID, S.Department_Name from R, S where R.Department_ID = S.Department_ID의 쿼리로 수행된다. 두 테이블에서 각 필드(field)는 4 바이트로 가정하였다. 따라서 R (또는 S) 테이블의 각 레코드의 크기는 8 바이트이다. 또한 페이지의 크기는 4KB로 가정하기 때문에, 총 512개의 레코드들이 한 페이지(page)에 저장된다. 그리고 메인 메모리와 하드디스크 간의 데이터 입출력은 페이지 단위로 수행된다고 가정한다. 조

1) 본 논문에서는 두 테이블을 R과 S로 칭한다.

2) 실제 상용 시스템과 비교하여, 실험에서 사용된 메인 메모리의 크기는 매우 작은 것을 알 수 있다. 본 논문에서는 시스템의 사양보다는 PRAM 기반의 시스템의 조인 알고리즘들의 내구성과 성능 비교에 초점을 맞추기 때문에, 메모리의 크기는 중요한 문제가 아니라고 판단하였다.

인 연산이 수행되고 나면, 조인 결과는 메인 메모리의 출력 버퍼 영역에 임시 저장되고, 버퍼가 풀(full)이 되면, 하드디스크로 플러시(flush)된다. 조인 결과의 각 레코드는 <Employee_ID, Department_ID, Department_Name>으로 구성되고, 크기는 12 바이트이므로, 한 페이지에 총 341개의 레코드가 저장된다.

이러한 PRAM 기반의 시스템에서 R 테이블[3]과 S 테이블은 각각 5백만 개와 5십만 개의 레코드를 가지고, Block Nested Loop Join, Sort-Merge Join, Grace Hash Join, Hybrid Hash Join 연산을 수행한다.

3.1 Block Nested Loop Join (BNL)

R 테이블은 블록(block) 단위로 분할한다. 편의상 한 블록의 크기를 한 페이지라고 가정하면, 한 블록 당, 512개의 레코드들이 포함된다. R 테이블의 첫 번째 블록(B_1^R)을 M_I 을 거쳐 M_R 에 저장한다. 마찬가지로 하드디스크로부터 S 테이블의 첫 번째 블록(B_1^S)을 M_I 을 거쳐 M_S 에 저장한다. 그리고 다음과 같은 nested loop 연산을 수행한다.

```
// r_i, r_j : 레코드
for i=1 to 512 in B_1^R
    for j=1 to 512 in B_1^S
        if (r_i ≡ r_j) then
            write (r_i, r_j) =< a_1, a_2, a_3 > to M_O
```

a_1, a_2, a_3은 각각 R.Employee_ID, S.Department_ID, S.Department_Name을 의미한다. M_O 의 공간이 모두 차면, 하드디스크에 저장된다. B_1^R 과 B_1^S 의 조인 연산이 끝나면, S 테이블의 두 번째 블록(B_2^S)을 M_I 를 거쳐 M_S 에 저장하고, 중첩 루프 조인(nested loop join)을 수행한다. 다음, B_1^R 과 S 테이블의 모든 블록에 대한 조인 연산이 끝나면, R 테이블에서 두 번째 블록(B_2^R)을 M_I 를 거쳐 M_R 에 저장하고, 중첩 루프 조인을 수행한다.

3.2 Sort-Merge Join (SM)

우선 R 테이블은 런(run) 단위로 분할된다. 최초 런의 크기는 페이지 사이즈(page size)이다. R 테이블의 모든 런 들을 각각 런 단위로 M_H 에 올려서 퀵 소트(quick sort)를 수행 후 다시 디스크에 저장한다. 이후 디스크에 있는 두 개의 런을 M_R 에 올려서 레코드 단위로 비교한 후 더 작은 쪽을 M_O 에 저장한다. M_O 가 가득 차게 되면 디스크에 저장한다. 이렇게 2개의 런을 디스크에 저장하게 되면 크기가 2배인 정렬된 하나의 런이 만들어진다. 이러한 방식으로 R 테이블에 대하여 외부정렬(external sort)을 수행 하면 디스크에는 글로벌 소팅(global sorting)된 R 테이블이 저장된다. 같은 방법으로 S 테이블에 있는 레코드들도 차례대로 정렬을 수행한 다음 하드디스크에 저장한다.

위의 과정을 확장하여 비슷한 맥락으로 이번에는 R 테이블과 S 테이블을 한 블록씩 M_R 과 M_S 에 올려서 비교 한다. 조인 조건을 만족하는 레코드들을 조인하여 M_O 에 저장한다. 이렇게 조인 작업까지 마치게 되면 디스크에는 최종적으로 R과 S 테이블에 대한 조인 조건을 만족하는 레코드들만 저장되게 된다.

3.3 Grace Hash Join (GH)와 Hybrid Hash Join (HH)[12]

첫 번째 단계(first step)에서 R 테이블은 블록(block) 단위로 분할한다. R 테이블의 첫 번째 블록(B_1^R)을 M_I에 넣는다. 그리고 M_I 의 내용을 읽어서 M_R 에 저장한다. M_R 이 다 차면, 1차 해시 함수(hash function)를 사용하여 각 레코드를 해당 버킷에 어사인(assign)한다[13].

> 1차 해시 함수: 알파벳의 i번째 문자는 정수 i로 나타낸다는 것을 가정한다. s는 길이가 n인 문자열을 나타내고 s[i]는 문자열의 i번째 byte를 표시할 때,
> $h_1(s) =$
> $s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \ldots + s[n-1]$

예를 들면, s = Department_ID. $h_1(s)$%(버킷수)를 사용하여 레코드가 저장될 버킷(bucket)이 결정된다.[4] 그러면 저장될 버킷에 상응하는 M_O 에 레코드가 저장된 후, M_O 가 가득차면 버킷에 저장된다. M_O 의 개수는 버킷의 개수와 동일하고 각각의 크기는 $\frac{4KB}{\text{버킷수}}$이다. 같은 방식으로 R 테이블의 다른 블록들을 해싱 하여, 레코드들을 버킷들에 저장하고 하드디스크에 저장한다. S 테이블의 모든 블록들도 동일한 과정을 거쳐 해싱 하고, 버킷들을 하드디스크에 저장한다. 이와 같이, 동일 해시 함수를 사용하여, R과 S 테이블에 있는 모든 레코드들을 해당 버킷들에 어사인하여, 하드디스크에 저장한다.

3) 균등 분포(uniform distribution) 보다는 실제 데이터의 분포에 가깝게 하기 위해, R 테이블의 레코드들은 Zipf 분포에 의해 생성된다고 가정한다.

4) 만일 버킷이 충분한 공간을 갖고 있지 않을 경우에는 분리 체인(separate chaining) 방식으로 해시 테이블을 구성한다. 예를 들면, 레코드가 버킷 b에 추가되어야 하는데 b가 이미 차있다면 시스템은 b을 위한 오버플로 버킷을 제공하고, 이 레코드를 오버플로 버킷에 삽입한다. 오버플로 버킷 또한 차게 되면 시스템은 또 다른 오버플로 버킷을 제공한다. 주어진 버킷의 모든 오버플로 버킷은 연결 리스트로 연결된다.

두 번째 단계(second step)에서는 S 테이블의 첫 번째 버킷($Bucket_1^S$)을 M_I에 넣는다. M_I의 내용을 읽어서, M_H에 해시 테이블을 생성한다. 이 때 사용되는 2차 해시 함수는 $h_2(s)=s[0]+s[1]+...+s[n-1]$로 정의하고, $h_2(s)$%(버킷수)에 의해 해시테이블에서 레코드가 저장될 버킷이 결정된다. 다음, $Bucket_1^R$을 M_I에 저장하고, 각 레코드를 $h_2(s)$을 통해 M_H에 해시 테이블의 버킷에 어사인 된다. 이때, 만일 특정 버킷에 3개의 레코드 $r_1=<2,*>\in Bucket_1^S$, $r_{15}=<*,4>\in Bucket_1^R$, $r_{100}=<*,2>\in Bucket_1^R$ 이 있다고 가정하자. r_{15}와 r_{100}의 두 번째 칼럼은 Department_ID인 반면에 r_1의 첫 번째 칼럼은 Department_ID이다. r_1과 r_{100}의 Department_ID는 같기 때문에, M_O에 $(r_1,r_{100})=<*,2,*>$은 저장되고, M_O가 가득 차거나, M_O가 다 차지 않더라도 버킷의 조인 연산이 끝날 때, M_O의 내용이 플러시 되어 하드디스크에 저장된다. $Bucket_1^R$과 $Bucket_1^S$의 조인 연산이 모두 끝나면, $Bucket_2^R$과 $Bucket_2^S$의 조인 연산이 차례로 수행된다.

Grace Hash Join과 Hybrid Hash Join의 알고리즘은 동일하다. 단, Hybrid Hash Join의 경우에는 첫 번째 단계에서 R과 S 테이블의 첫 번째 버킷은 하드디스크에 저장하지 않고, $h_2(s)$를 사용하여 M_H에 해시 테이블을 만들고, 조인 연산을 수행한다. 반면에 나머지 버킷들은 Grace Hash Join과 마찬가지로 하드디스크에 저장하고, 두 번째 단계에서 $h_2(s)$를 사용하여 조인 연산을 수행한다.

4. 실 험

4.1 실험환경

표 2에서는 실험을 진행하기 위하여 모델링한 결과를 기술하였다. 하드웨어, 테이블, 해시 관련 파라미터들을 기술하였다. 메인 메모리는 총 32KB로 출력버퍼는 4KB를 할당하고, 입력버퍼는 블록 사이즈(block size)×8 인 4KB를 할당하여 최대 8블록을 멀티 액세스 가능하다[14]. 워크 스페이스(work space)인 R, S 파트에 각 8KB를 할당한다. 남은 8KB는 해시와 정렬에 사용된다. R 테이블의 참조키는 지프(Zipf) 분포로 생성 되었는데 데이터의 발생 빈도가 뒤로 갈수록 적게 나타나는 분포이다. S 테이블의 값 범위(value range)는 0에서 10,000 사이의 무작위 값을 추출하였다. 버킷의 개수 |B|는 10개로 설정하며, 버킷 하나의 크기는 4096 바이트이다.

기존의 연구에서는 버퍼 공간(buffer pool)을 만들고 이 영역에 할당되는 메모리의 구성에 따라서 입력 버퍼(input buffer), 출력 버퍼(output buffer), 워크 스페이스(R, S, Hash 등)의 크기와 형태가 달라진다. 하지만 본 논문에서는 각각의 영역을 고정영역으로 할당한 후 각 영역별 읽기(read)/쓰기(write)을 분석하는 관점으로 접근하고자 한다. 이를 통해 각 각의 조인 알고리즘 마다 영역별 읽기/쓰기 연산의 분포를 파악할 수 있다.

표 2 실험환경 파라미터
Table 2 Experimental Environment Parameter

Hardware modeling	Value
Page size	4096 bytes
Block size	512 bytes
Disk size	4GB
PRAM word read latency	0.000005 ms
PRAM word write latency	0.000062 ms
Main memory size	32KB
- Input buffer area	- 4KB
- Output buffer area	- 4KB
- R table area	- 8KB
- S table area	- 8KB
- Other memory area	- 8KB
- Sort memory or Hash memory	- 4KB or 8KB
Table modeling	**Value**
R table <Department_ID, Department_Name>	2 columns, 8 bytes 500,000 records uniform distribution
S table <Employee_ID, Department_ID>	2 columns, 8 bytes 5,000,000 records Zipf's distribution
Output table	3 columns, 12 bytes
Hash modeling	**Value**
Hash range (Number of buckets)	10
Bucket size	4096 bytes

4.2 실험결과

4.2.1 메인메모리 Write 연산 횟수

표 2에 기술된 실험환경의 순서대로 입력버퍼, 출력버퍼, R 테이블 영역, S 테이블 영역, 기타영역 순으로 나타나고 있다. 차지하는 메모리 크기 또한 기술한 내용과 일치한다.

그림 1을 보면 입력 버퍼의 사용량이 굉장히 많은데 이것은 BNL 조인의 방식이 이중루프를 수행하는 방식이므로 프로빙 테이블(probing table)이 반복적으로 읽혀진 결과이다. 출력 버퍼의 경우는 조인 수행 과정에서 버퍼가 가득 차면 플러시(flush) 하게 된다. 그러나 연산의 수행 단위인 한 블록을 처리하고 나면 새로운 블록을 불러올 때 까지 기다림이 발생하는데 출력버퍼를

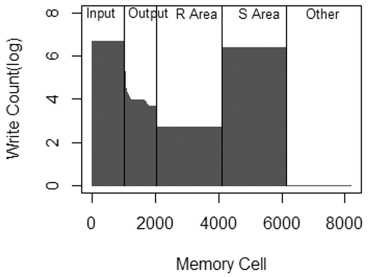

그림 1 블록 네스티드 루프 조인 메모리 쓰기 횟수
Fig. 1 BNL Join Memory Write Count

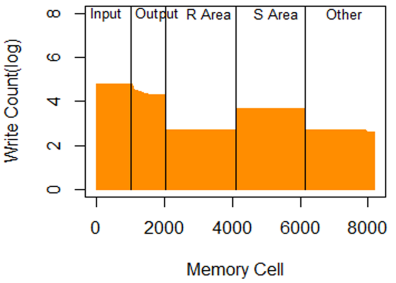

그림 3 그레이스 해시 조인 메모리 쓰기 횟수
Fig. 3 GH Memory Write Count

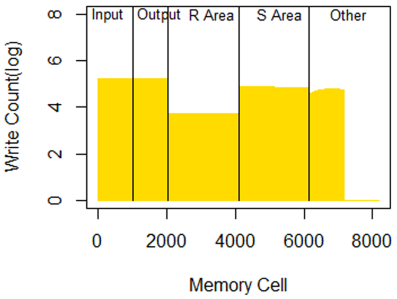

그림 2 소트 머지 조인 메모리 쓰기 횟수
Fig. 2 SM Join Memory Write Count

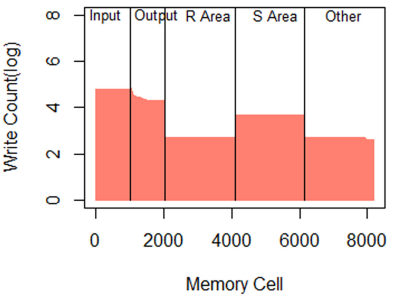

그림 4 하이브리드 해시 조인 메모리 쓰기 횟수
Fig. 4 HH Memory Write Count

플러시하지 않는다면 자원의 독점을 의미하므로 본 실험에서는 블록단위마다 버퍼를 플러시 하도록 설계하였다. 또한 가득 차지 않은 버퍼를 플러시 하는 경우가 반드시 발생하게 되며, 그러한 비효율이 누적된 그래프가 그림 1과 같다. 메모리의 R 테이블 영역의 사용량이 굉장히 많은데 이것은 입력버퍼와 마찬가지로 R 테이블의 블록을 반복적으로 적재하기 때문이며 메모리 영역의 크기가 입력버퍼보다 크기 때문에 입력버퍼 보다는 사용량이 적게 나타난 것이다. S 테이블 영역의 경우 반복 적재가 일어나지 않으므로 사용량이 굉장히 적은 모습이다. 끝으로 BNL 조인의 경우 해시테이블 사용 혹은 정렬 작업 수행을 하지 않으므로 기타영역은 사용하지 않는다.

그림 2의 SM 조인의 경우 BNL 조인과의 차이점은 각 테이블을 외부정렬 한 후에 조인을 수행한다는 점이다. 이 과정에서 디스크 입출력(I/O)가 많이 발생하여 출력 버퍼의 사용량이 다른 조인들보다 다소 높게 나타난다. 하지만 이중 루프 방식의 비효율적인 비교연산을 수행하지 않기 때문에 BNL에 비해서 입력 버퍼의 사용량이 훨씬 적은 모습이다. 전체적인 쓰기 연산 횟수와 분포면에서 BNL 보다 상당히 좋아진 모습을 볼 수 있다.

그림 3의 GH 조인은 해시 테이블이라는 추가적인 자원을 사용하여 조인을 수행하므로 좀 더 효율적인 메모리 사용이 가능해 전체적인 쓰기 연산 분포가 고르고 연산 횟수가 적게 나타난다. 이중루프를 수행하는 방식이 아니라 R과 S 테이블을 각각 풀 테이블 스캔 하는 방식이므로 입력 버퍼의 사용량이 현저히 줄어들었다. 출력 버퍼의 경우는 파티션 단계에서 동일한 해시함수를 사용해 버킷 페어를 만들 때 버킷을 적재할 때 버퍼가 가득차지 않고 플러시 되는 만큼 불균형이 나타난다. R, S 테이블 영역의 경우 두 번의 단계에서 모두 사용되지만 앞서 기술한 바와 같이 이중루프 방식이 아닌 스캐닝 방식이므로 쓰기 연산이 지수 적으로 상승하지 않는다. 끝으로 해시 메모리 영역은 8KB로 할당된 해시테이블에 레코드를 삽입 할 때마다 쓰기 연산이 발생하며, 블록 하나를 삽입 할 때마다 사용되지 않는 부분이 발생할 수 있지만 비교적 고른 사용 분포를 보인다.

그림 4의 HH 조인과 GH 조인 알고리즘의 차이는 파티션 단계에서 드라이빙 테이블(driving table)의 0번 버킷을 디스크에 저장하지 않고 곧바로 해시테이블을 생성하고, 프로빙 테이블(probing table)의 0번 버킷도 디스크에 저장하지 않고 해시테이블을 조회하여 바로 조인하는 부분이다. 이러한 방식을 사용하면 GH 조인보다 디스크 입출력을 줄일 수 있다. 그래프를 로그 스케일(log scale)로 봤을 때 두 그래프의 가시적인 차이는 나타나지 않지만 입출력 버퍼의 사용량이 약간씩 줄어들게 된다.

4.2.2 전체 메모리 사용 횟수

그림 5에서 BNL 조인은 메모리 읽기 연산이 유독 많은데 이것은 위에 기술한 것처럼 이중루프를 수행하는 과정 때문에 나타나는 현상이다. 즉 읽기 연산 대부분은 R 테이블의 블록을 반복적으로 참조하는 연산이다. SM 조인의 경우 읽기 연산이 가장 적게 나타난다. 이것은 R, S 테이블을 이중루프 방식으로 반복적으로 불러들이는 BNL 조인과 달리 각 테이블을 한번 씩 읽어 들이는 방식을 취하고 있으며 조인 단계에서도 각 테이블을 한 번 씩 스캔하는 것만으로 조인이 완료되기 때문이다. GH 조인은 쓰기 연산에서 BNL, SM 조인과 확연한 차이를 보이는데 그 이유는 최초 파티션 단계에서의 메모리 쓰기 연산과 해시 테이블을 생성하는 과정에서의 쓰기 연산이 있을 뿐 동일한 데이터를 반복적으로 쓰기 연산을 할 필요가 없기 때문이다. 반면에 읽기 연산의 경우 파티션 단계에서는 한 번의 테이블 스캔이 발생하지만 머지 단계에서 해시테이블을 반복적으로 참조하는 구조이므로 SM 조인의 읽기 연산과 약간의 차이를 보일 뿐이다. HH 조인은 GH 조인에 비해서 입출력 버퍼의사용량이 줄어듦에 따라 전체 읽기와 쓰기 횟수가 약간씩 줄어든다.

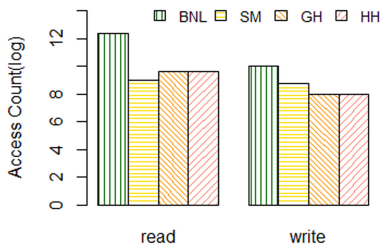

그림 5 전체 메모리 사용 횟수

Fig. 5 Total Memory Access Count

4.2.3 전체 메모리 사용 시간

BNL 조인과 SM 조인을 비교해 보면 BNL 조인은 그림 5에서 압도적인 읽기 횟수를 보인다는 것을 확인했으므로 그림 6의 읽기 연산에 걸린 시간(read time) 또한 굉장히 오래 걸리는 것을 알 수 있다. 쓰기 연산도 가장 오랜 시간이 걸렸지만 읽기 연산의 비중으로 전체

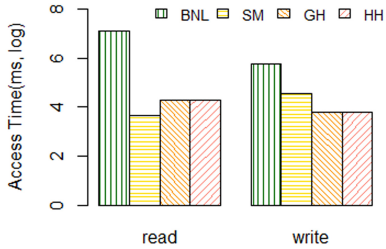

그림 6 전체 메모리 사용 시간

Fig. 6 Total Memory Access Time

속도가 상당히 느려짐을 알 수 있다. SM 조인의 경우는 알고리즘의 효율성이 개선됨에 따라 읽기 연산에 걸린 시간의 급격한 감소를 볼 수 있다. GH 조인의 경우 메모리 액세스 측면만 놓고 본다면 상당한 성능을 보여주지만, 파티션 단계를 수행하기 위한 추가적인 디스크 입출력이 발생하게 되므로 디스크 입출력 시간이 SM 조인보다 더 오래 걸리게 된다. 이러한 디스크 입출력을 줄이기 위한 방안으로 Hybrid Hash Join 기법을 사용할 수 있다. HH 조인은 위에서 기술한 것처럼 GH 조인에 비해서 입출력 버퍼의사용량이 줄어든 만큼 읽기와 쓰기 연산의 수행속도도 약간씩 향상된 모습이다.

4.3 결과비교

표 3의 실험결과 비교표를 보면 타 조인 알고리즘에 비해서 Hash 조인 알고리즘이 메모리 쓰기(memory write)과 균등분배 측면에서 성능 상 뛰어나다는 것을 알 수 있다. Hash 조인이 Sort-Merge 조인보다는 읽기 연산이 걸린 시간(read time)이 오래 걸리지만 전체 읽기쓰기 시간과 균등분배 정도를 고려하면 역시 Hash 조인의 성능이 더 우수하다고 할 수 있다. Hybrid Hash 조인은 Grace Hash 조인에 비해서 극명한 차이는 아니지만 메모리 쓰기횟수, 평균, 표준편차, 쓰기 연산이 걸린 시간(write time)이 0.62% 줄었고, 읽기 연산이 걸린 시간(read time)이 0.89% 줄어 균등분배 및 읽기쓰기 측면에서 약간의 성능 향상이 있다.

4.4 세부분포

본 논문에서 연구한 Hybrid Hash 조인은 Grace Hash 조인을 개선시키기 위해 고안해낸 알고리즘 방식이며

표 3 실험결과 비교

Table 3 Comparison of Experimental Result

		Block Nested	Sort-Merge	Grace Hash	Hybrid Hash
Memory Write Count		9,797,000,000	564,551,436	99,999,600	99,377,152
Write Count Load Balancing Indicator	Mean	1,594,563.8	78,759.97	12,206.98	12,131.0
	Standard Deviation	1,777,922.92	63,441.05	20,134.25	20,009.49
Performance	Read Time (ms)	12,536,446.21	4,506.7	20,345.27	20,163.3
	Write Time (ms)	607,414.06	35,002.19	6,200.0	6,161.38

현재 까지도 다양한 형태로 응용된 Hybrid Hash 조인이 연구 중에 있다. 따라서 성능이 좋으며 앞으로 알고리즘 개선의 여지가 가장 큰 Hybrid Hash 조인의 분포를 세분화하여 분석해 본다.

4.4.1 입력버퍼 세부분포

메모리 셀의 후반부로 갈수록 사용량이 줄어드는 것을 볼 수 있다. 파티션 단계에서는 사실상 고른 분포를 유지하지만 머지 단계에서 프로빙 테이블(R 테이블)을 가져올 때 버킷마다 들어있는 레코드의 수가 균일하지 않기 때문에 입력 버퍼의 분포가 그림 7과 같이 나타나게 된다.

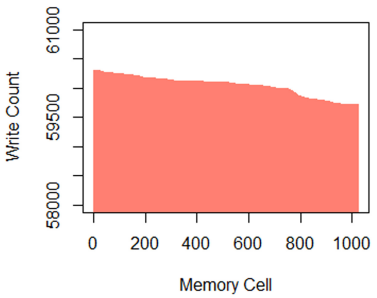

그림 7 하이브리드 해시 조인 입력버퍼 쓰기 횟수
Fig. 7 HH Input Buffer Write Count

4.4.2 출력버퍼 세부분포

출력버퍼의 경우 사용량의 격차가 큰 편인데 이러한 차이가 발생하는 주된 원인은 머지 단계에서 버퍼가 가득 차지 않더라도 한 블록을 처리할 때마다 버퍼를 플러시 하므로 메모리 셀의 뒷부분을 사용하지 않는 경우가 누적되기 때문이다(그림 8 참조). 이러한 문제를 해결하기 위해서 버퍼 Flush 시점을 늦추는 방법이 있다. 그러나 이러한 방법을 사용하게 되면 한 블록의 연산을 마치더라도 버퍼에는 저장되지 않은 데이터가 남아있게 되므로 다른 프로세스가 버퍼를 사용하지 못하는 독점 현상이 발생할 수 있다. 또는 버퍼의 내용을 임시저장 해야 하는데 이것은 또 다른 입출력 오버헤드를 의미한다.

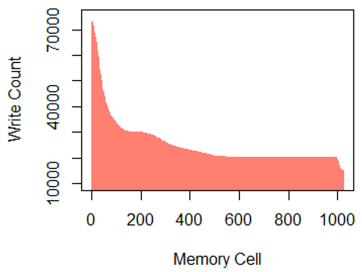

그림 8 하이브리드 해시 조인 출력버퍼 쓰기 횟수
Fig. 8 HH Output Buffer Write Count

4.4.3 R 테이블 영역 세부분포

HH 조인은 R 테이블 영역을 사용하는 경우가 파티션을 위해 최초에 테이블을 스캔할 때뿐이다. 따라서 그림 9와 같이 고른 분포가 나타나며 사용량은 테이블의 크기에 비례한다.

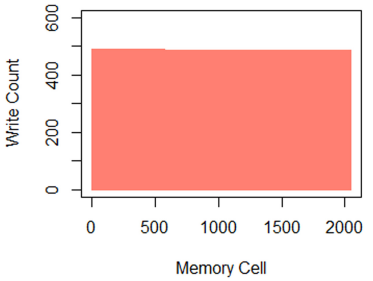

그림 9 하이브리드 해시 조인 R 테이블 영역 쓰기 횟수
Fig. 9 HH R Table Area Write Count

4.4.4 S 테이블 영역 세부분포

S 테이블 영역도 R 테이블 영역과 같은 결과이다. 테이블 사이즈에 따라서 사용량도 적게 나타난다. (그림 10 참조)

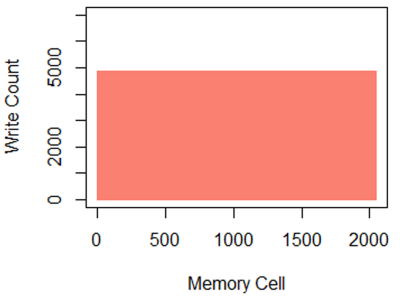

그림 10 하이브리드 해시 조인 S 테이블 영역 쓰기 횟수
Fig. 10 HH S Table Area Write Count

4.4.5 기타 영역 세부분포

해시 테이블은 드라이빙 테이블(driving table)의 한 버킷을 담을 때 마다 쓰기 연산이 발생한다. 이때 버킷의 레코드가 해시테이블 영역을 가득 채우지 못한다면 해시테이블 영역을 전부 사용하지 못하게 되고 이러한 경우가 누적 되서 그림 11과 같이 뒤쪽 메모리 셀을 적게 사용하는 분포가 나타난다.

4.5 결과분석

여타 조인에 비해서 Hybrid Hash Join의 경우가 메모리 Access 측면에서 장점을 보이는 것은 사실이지만 조금 더 메모리 사용 분포를 고르게 개선시킬 여지가 있다. 첫 번째 방법으로는 버퍼 Flush 시점을 바꾸는 방법이다. 버퍼를 덜 자주 비우게 되면서 버퍼에 대한

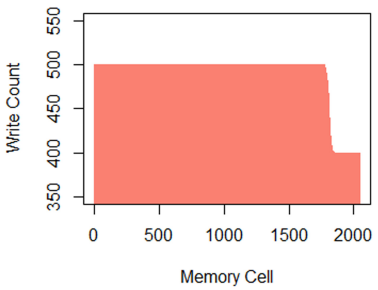

그림 11 하이브리드 해시 조인 해시메모리 쓰기 횟수
Fig. 11 HH Hash Memory Write Count

Write 연산이 조금 더 고르게 나타날 수 있다. 두 번째 방법으로는 더욱 효율적인 조인 알고리즘을 만드는 것이다. Hash Join을 기반으로 Hybrid Hash Join보다 개선된 알고리즘 사용하면 메모리의 효율적인 사용이 가능하다.

5. 결 론

본 논문에서는 PRAM을 메인 메모리로 사용하는 시스템에서 기존에 널리 사용되는 Block Nested Loop Join, Sort-Merge Join, Grace Hash Join, Hybrid Hash Join을 적용했을 때 발생하는 내구성과 성능 이슈를 실험을 통해 비교, 분석하였다. 그리고 기존의 조인 알고리즘[15-18]들을 PRAM에 맞게 재설계해야 되는 당위성을 고찰하였다. 기존의 PRAM 연구들이 주로 DRAM을 대체하기 위한 하드웨어적인 기법을 제안한 것에 비해, 본 연구에서는 조인 알고리즘과 같이 많이 사용되는 어플리케이션이 소프트웨어적으로 어떤 문제를 발생하는지 규명하는 첫 번째 시도였고, PRAM 기반의 시스템을 모델링하고 시뮬레이터를 구현한 것에 연구의 큰 의의를 둘 수 있다.

향후 연구로는 기존 연구를 확장하여 PRAM 기반의 시스템을 테스트베드로 구축하고, 실제 성능 테스트를 수행할 예정이다. 또한 기존 조인 알고리즘을 PRAM에 맞게 재설계하여, 효용성을 입증하는 것이 주요한 작업이 될 예정이다.

References

[1] Hana Institute of Finance, Outlook for non-volatile memory devices and solid state drives, 2009.
[2] M. Qureshi, V. Srinivasan and J. Rivers, Scalable high performance main memory system using phase-change memory technology, *Proc. of the 36th Annual International Symposium on Computer Architecture*, pp. 24-33, 2009.
[3] D. Woodhouse, JFFS: The journalling flash file system, *Proc. of the Linux Symposium*, Jul. 2001.
[4] D. Kang, D. Jung, J. Kang and J. Kim, μ-tree: An ordered index structure for NAND flash memory, *Proc. of the 7th ACM/IEEE International Conference on Embedded Software*, pp. 144-153, 2007.
[5] Y. Li, B. He, J. Yang, Q. Luo and K. Yi, Tree indexing on solid state drives, *VLDB Journal*, Vol. 3, No. 1, pp. 1195-1206, 2010.
[6] ZDNet Korea, NAND Flash market, 18% growth this year, 2011, Available from http://www.zdnet.co.kr/news/news_view.asp?artice_id=20110120190952
[7] B. Lee, E. Ipek, O. Mutlu and D. Burger, Architecting phase change memory as a scalable dram alternative, *Proc. of the 36th Annual International Symposium on Computer Architecture*, pp. 2-13, 2009.
[8] S. Cho and H. Lee, Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance, *Proc. of the 42nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 42)*, pp. 347-357, 2009.
[9] J. Kim, H, Lee, S. Choi and K. Bahng, A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems, *Proc. of the 8th ACM International Conference on Embedded Software*, pp. 31-40, New York, NY, USA, 2008.
[10] S. Nath and A. Kansal, FlashDB: Dynamic self-tuning database for NAND flash, *Proc. of the 6th International Conference on Information Processing in Sensor Networks*, pp. 410-419, 2007.
[11] J. Do and J. Patel, Join processing for Flash SSDs: Remembering past lessons, *Proc. of 5th International Workshop on Data Management on New Hardware*, Providence, Rhode-Island, USA, Jun. 2009.
[12] J. Patel, M. Carey and M. Vernon, Accurate modeling of the hybrid hash join algorithm, ACM SIGMETRICS Conference, Nashvile, 1994.
[13] A. Silberschatz, H. Korth and S. Sudarshan, Database system concepts, McGraw-Hill Education, 6th Edition, 2011.
[14] L. Haas, M. Carey, M. Livny and A. Shukla, Seeking the truth about ad hoc join costs, *VLDB Journal*, Vol. 6, No. 3, pp. 241-256, 1993.
[15] J. Park, S. Park, S. Lee and C. Park, Performance evaluation of hash join algorithm on flash memory SSDs, *Journal of KIISE: Computing Practices and Letters*, Vol. 16, No. 11, pp. 1031-1040, 2010.
[16] Y. Kim, S. Kim, Y. Park, S. Jin and K. Rim, A load balancing hash join algorithm for data skew in shared-nothing parallel database system, *Journal of KIISE*, Vol. 22, No. 8, pp. 1174-1182, 1995.
[17] Y. Shim and J. Lee, A skewed data handling method using spatial hash join algorithm, *Journal of KIISE*, Vol. 31, No. 1(B), pp. 19-21, 2004.
[18] C. Kim and H. Cho, Implementation of parallel hash join algorithms in a database sharing system, *Journal of KIISE*, Vol. 29, No. 1B, pp. 43-45, 2002.

최 용 성

2008년~현재 경기대 컴퓨터과학과 4학년. 관심분야는 비휘발성 메모리 기반 데이터베이스 시스템

온 병 원

2007년 펜실베이니아주립대 컴퓨터공학과 박사. 2008년 브리티시컬럼비아대 포스닥 연구원. 2010년 일리노이대 ADSC 선임연구원. 2011년 차세대융합기술연구원 선임연구원. 2014년 군산대 통계컴퓨터과학과 조교수. 관심분야는 데이터 마이닝, 데이터베이스, 정보검색, 빅데이터

최 규 상

2005년 펜실베이니아주립대 컴퓨터공학과 박사. 2006년~2009년 삼성종합기술원 전문연구원. 2009년~현재 영남대학교 정보통신공학과 부교수. 관심분야는 임베디드 시스템, 스토리지 시스템

이 인 규

2007년 펜실베이니아주립대 컴퓨터공학과 박사. 2007년~2013년 미국 트로이주립대 조교수. 2013년~2014년 차세대융합기술연구원 스마트그리드연구센터 책임연구원. 2015년~현재 미국 트로이주립대 부교수. 관심분야는 데이터 마이닝, 소셜 네트워크 분석 및 비즈니스 애널리틱스