요약

본 논문에서는 관계 데이타베이스 시스템의 그룹연산을 위한 내부 구현 알고리즘 중 중첩루프(Nested-loops) 알고리즘, 정렬-합병(Sort-merge) 알고리즘, 해쉬(hash) 알고리즘을 직접 구현하고, 이를 메모리의 크기와 입력 파일의 튜플의 수를 변화시켜가며 성능을 비교, 분석하였다. 이를 통하여 대용량의 메인 메모리를 가지는 현대의 데이타베이스 환경에 비추어 각 알고리즘의 성능과 효용성을 평가하였다. 실험 결과 각 경우에 대하여 평균적으로 정렬-합병(sort-merge) 알고리즘이 가장 뛰어난 성능을 보여주었고, 해쉬 알고리즘의 경우 정렬-합병 알고리즘과 거의 비슷한 수준의 성능을 보여 주었다. 중첩루프 알고리즘의 경우 튜플의 수가 커질 경우 수행 시간이 너무 길어져 그룹 연산을 위한 적합한 알고리즘이 될 수 없음을 알 수 있었다. 정렬-합병 알고리즘이나 해쉬 알고리즘의 경우 메인 메모리를 효율적으로 사용할 수 있으므로 그룹 연산의 구현에 효과적으로 이용할 수 있는 알고리즘임을 알 수 있었다.

제 1장. 서론

데이타베이스 처리 환경에서 컴퓨터의 하드웨어적인 기술이 크게 발달함에 따라 마이크로 프로세서의 속도는 크게 향상되었으나, 낮은 디스크의 효율이 데이타베이스 접근의 주요 문제점으로 나타나고 있다. 대량의 멀티미디어 데이타, 즉 이미지, 비디오, 오디오 등의 데이타를 처리하기 위하여서는 이러한 데이타베이스 접근에 관한 문제점이 더욱 부각되고 있다. 이와 함께 메인 메모리 가격의 하락과 직접도의 향상으로 인하여 과거의 보조 저장장치의 용량을 뛰어넘는 대량의 메인 메모리를 가진 시스템들이 널리 사용되고 있다. 이에 따라 데이타베이스의 조작을 위한 오퍼레이션의 내부 처리를 위해서 과거에는 사용할 수 없던 알고리즘들도 새로이 사용되어지고 있다.

이미 80년대 중반부터 이러한 문제에 대한 연구가 활발히 진행되어 지고 있다. 특히 소트와 해쉬를 이용한 여러 기법들이 대용량의 메인 메모리를 이용한 알고리즘으로 연구되고, 이미 사용되어져 왔다.

본 논문에서는 이러한 데이타베이스의 여러 응용 중에서도 한 릴레이션을 가지고 Grouping을 수행할 때의 수행 알고리즘을 직접 구현하고 그 성능을 비교 평가하였다. 이는 관계 데이타베이스를 위한 연산 중 Join에 관한 연구와 유사한 점이 있다. 그룹으로 묶는 연산이 마치 하나의 파일에서 같은 파일로의 Join과 유사하기 때문이다. 따라서 알고리즘도 관계 데이타베이스 연산 중 Join을 위한 구현 알고리즘과 유사한 알고리즘들로 구현되어질 수 있다. Join연산은 튜플전체를 대상으로 수행되어야 하므로 실제 수행 성능의 개선 여지가 가장 많은 연산 중하나이다. 그룹연산도 마찬가지로 알고리즘에 따라 수행 성능에 많은 차이가 있을 수 있다. 이 논문에서는 이러한 각 알고리즘간의 성능의 차이를 튜플의 수와 메인 메모리의 사용 가능한 크기에 따라 비교 분석해 보았다.

본 논문에서는 임의의 파일을 하나 생성하여 이를 선택된 필드에 의하여 그룹연산을 하고 이를 카운트하는 작업을 중첩루프(nested-loops) 알고리즘, 외부 합병 정렬, 해쉬 알고리즘을 이용하여 직접 수행하고 이에 걸리는 시간을 측정하여 그 성능을 비교 분석하였다. 이를 튜플의 수를 변화시켜가며, 메인 메모리의 크기를 변화시켜가며 각 경우의 성능을 분석하였다.

본 논문은 각 알고리즘을 실제 구현하고 알고리즘의 수행성능을 비교하고 분석한다. 이를 위한 본 논문의 구성은 다음과 같다. 2장에서는 관련연구들을 조인 algorithm을 위주로 소개하고, 3장에서는 실제 알고리즘의 구현 내용을 제시한다.

4장에서는 3장에서 구현된 프로그램에 의해 실제 실험한 자료에 의해 메모리 크기에 따른 세 알고리즘의 수행성능과 튜플의 크기에 따른 세 알고리즘의 수행성능에 대하여 비교 분석하였다. 마지막으로 5장에서는 이 논문의 결론과 향후 연구과제에 대하여 언급한다.

제 2장. 관련연구

지금까지 데이타베이스 조작 연산의 내부 구현을 위한 여러 알고리즘이 소개되어져왔고, 많은 연구에서 분석되어져 왔다. 이러한 분석은 특히 Join알고리즘에서 많이 이루어져왔다. 이는 Join연산이 대단히 복잡한 연산으로서 데이타베이스의 응답시간 단축의 주요 변수이기 때문이다. 이러한 Join연산의 내부 구현에 관하여 살펴보자.

관계 데이타베이스를 위한 Join알고리즘들로서는 대표적으로 sort-merge[1] join algorithm, Simple Hash Join Algorithm, Grace Hash Join Algorithm[2], Hybrid-Hash Join Algorithm 등이 있다. 참고문헌[3]에서는 이러한 4가지 Join알고리즘의 성능을 비교하였다. 이 논문은 각 Join연산 알고리즘의 성능을 메모리의 크기와 튜플의 수의 비에 의해 분석하였다. 메인 메모리의 사용이 가능한 페이지 수를 M이라 하고, 릴레이션의 튜플 수를 R이라고 하고, universal "fudge" factor를 F라 할 때, $\frac{|M|}{|R|*F}$에 따라 각 알고리즘을 비교한 결과, Simple-hash알고리즘은 이 값에 많은 변화를 보였고, Sort-merge, Grace-hash, Hybrid-hash 알고리즘은 그리 크게 변하지 않는 우수한 성능을 보였다. 특이할 점은 이 비율의 값이 커짐에 따라, 즉 메인 메모리가 상대적으로 커지면 해쉬 기법의 알고리즘들이 전반적으로 우수한 성능을 보였다.

현재까지 알려진 대표적인 Join의 구현 알고리즘에 대하여 [4]를 참고하여 살펴보자.

2.1 중첩 루프(Nested Loops) 조인 알고리즘

중첩루프 조인 알고리즘은 가장 간단한 조인방법이다. 한 릴레이션에 속한 모든 튜플을 다른 릴레이션의 모든 튜플과 일일이 읽고 비교하는 알고리즘이다. $R \bowtie_{r(a)\,\theta\,s(b)} S$의 경우 다음과 같이 수행된다.

```
for each tuple s do
 { for each tuple r do
       { if r(a) θ s(b) then
       concatenate r and s
       place in relation Q}}
```

일반적으로는 Nested Loops Join Algorithm은 nested block join기법으로 구현된다. 즉, 일반적으로 튜플 하나 하나를 다루기보다는 하나의 블록 단위로 검사를 하게 된다. 이 구현 방법은 한번에 내부 릴레이션의 한 블록을 읽어서 조인을 한다. 내부 릴레이션의 한 블록내의 모든 튜플은 외부 릴레이션의 모든 블록의 튜플과 조인된다. 이 과정이 내부 릴레이션의 모든 블록에 계속된다. 얼마나 많은 I/O 작업을 줄일 수 있느냐는 사용 가능한 메인 메모리의 크기에 따라 결정된다. 보다 개선된 방법으로는 내부 릴레이션을 'rocking'하는 방법이 소개되어 있다. 위에 소개된 알고리즘에 따르면 두 릴레이션의 모든 튜플을 검사해야 하기 때문에 당연히 $O(n \times m)$의 수행시간을 따른다. block-oriented된 구현방법을 사용하면 I/O의 오버헤드를 더욱 최적화 할 수 있다.

이 알고리즘은 일반적으로 모든 튜플을 검사해야 하기 때문에 실제로는 조인연산을 위하여 잘 사용되지는 않는다. 따라서 실제 사용되는 비율은 높지 않다. 하지만, 구현이 다른 알고리즘에 비해 간단하므로 데이타베이스 머신을 위한 하드웨어 구현에는 널리 이용되고 있다[5]. 이 알고리즘은 병렬적으로 수행되어 질 수 있기 때문에, 병렬 수행 알고리즘으로 구현되었을 때 더욱 우수한 성능을 보인다. 이 외에도 이 알고리즘은 DBGraph라 불리는 main memory database를 위한 모델로도 사용된다.

2.2 정렬-합병(Sort-merge) 조인 알고리즘

정렬-합병(Sort-merge) 조인 알고리즘은 두 단계로 나누어 진행된다. 우선 두 릴레이션을 선택된 애트리뷰트에 따라서 소트를 한다. 그 다음 두 릴레이션을 소팅된 순서에 따라 읽어 가면서 조인 조건을 만족하는 튜플을 찾아 하나의 릴레이션으로 합친다. 이 알고리즘으로 동일조인을 수행하는 가장 간단한 알고리즘은 다음과 같다.

```
Stage 1 : Sort process
sort R on r(a);
sort S on s(b);

Stage 2 : Merge process
read first tuple from R;
read first tuple from S;
for each tuple r do
 {while s(b)<r(a)
```

```
        read next tuple from S;
        if r(a) = s(b) then
              join r and s
              place in output relation Q};
```

만일 조인 attributes가 key attributes가 아니면 같은 값을 가진 튜플이 존재 할 수 있으므로 위의 알고리즘은 약간의 수정이 있어야 한다.

이 알고리즘의 수행시간은 사용되는 소트와 합병의 알고리즘에 달려있다. 만일 릴레이션이 이미 소트되어 있다면 조인의 전체 수행시간은 오로지 두 릴레이션을 합병하는데 좌우된다. 하지만, 일반적으로 이 알고리즘은 전체적으로 소트를 하는데 걸리는 시간에 의해 전체 수행 시간이 결정된다. 일반적으로 소트에 걸리는 시간은 $O(n \log n)$이다. 여기서 n이란 릴레이션의 cardinality이다.

만일 조인 attributes에 인덱스가 없고, 특별히 알려진 특성이 없으며 기본적으로 선택된 부분 조인 기법이 없다면, 이 알고리즘이 가장 널리 선택되는 방법이다. 일반적으로 하드웨어적인 sorter의 도움을 받을 수 있으므로, 이 알고리즘은 하드웨어적인 구현도 간단하다. VERSO와 같은 몇 몇 데이타베이스 시스템에서는 이 알고리즘을 기본 조인 연산 방법으로 사용한다.

2.3 해쉬 조인 알고리즘

Sort-merge join algorithm의 성패여부는 얼마나 튜플간의 비교횟수를 줄이느냐에 달려있다. 해쉬를 이용한 기법은 이러한 방법에 대한 하나의 다른 대안이 될 수 있다. 어떠한 한 hash value를 가지는 튜플은 다른 hash value를 가지는 튜플과 join될 수는 없다. Hash join method는 이러한 원리를 응용한 것이다. hash join method는 우선 첫 번째 릴레이션의 튜플들을 두 번째 릴레이션과 join될 수 있도록 분리시켜 놓는다. 이렇게 하면 두 번째 릴레이션의 튜플들은 join을 위하여 첫 번째 릴레이션의 튜플들 중 제한된 일부의 집합만을 가지고 비교 join할 수 있다.

실제로 이러한 hash join method을 많은 데이타베이스 시스템에서 사용하고 있다. 이 중 몇 가지 방법을 살펴보면 다음과 같다.

2.3.1 Simple hash join method

simple hash join method는 hash value에 따라 전체 튜플이나 혹은 튜플 아이디

가 기록되는 튜플 테이블을 고른다. 이 방법은 hash함수의 효율성에 따라 알고리즘의 성능이 좌우된다. $R \bowtie_{r(a)\theta s(b)} S$의 경우 기본적인 알고리즘을 살펴보면 다음과 같다.

```
for each tuple s in S do
 { hash on join attributes s(b)
 place tuples in hash table based on hash values
 };

for each tuple r do
 { hash on join attributes r(a)
 if r hashes to a noempty bucket of hash table for S
 then {
        if r θ-matches any s in bucket
        concatenate r and s
        place in relation Q }};
```

이 알고리즘은 각 릴레이션이 한번만 읽혀지면 되므로 알고리즘의 Complexity는 단지 $O(n+m)$이다. 이 알고리즘의 수행 성능은 hash function의 효율성에 가장 연관이 크다. hash function이 우수하면 hash collision도 크게 줄어들 수 있기 때문이다. 이 방법은 간단하면서도 성능이 비교적 우수한 알고리즘으로 알려져 있다. 그러므로 하드웨어 hash unit를 제작하기 위한 방법으로도 많이 사용되고 있다.

2.3.2 Hash-partitioned joins

이 알고리즘은 hash를 이용한 Join을 위한 divide-conquer approach이다. 이 방법은 다른 방법과 달리 여러 이점을 가진다. 단지 성능 향상의 이점 외에도 각 partition이 병렬적으로 수행할 수 있는 이점도 있다. hash function은 하나의 split function의 역할을 한다. 이러한 방법으로는 Simple hash-partitioned join method, GRACE hash join method, Hybrid hash method, Hashed loops join등이 있다.

2.3.2.1 GRACE hash join method

GRACE hash join method은 join을 효과적으로 수행하기 위한 동적 클러스터링 기법을 사용하는 방법이다. $R \bowtie_{r(a)\theta s(b)} S$의 경우 기본적인 알고리즘은 다음과 같다.

```
for each tuple r do
 { hash the join attributes r(a)
 place tuple in appropriate output buffer  Ri};
flush all output buffers to disk;

for each tuple s do
 { hash the join attribute s(b)
 place tuple in appropriate output buffer  Si};
flush all output buffers to disk;

for i=1,2,...,N do
 { for  Ri do
       { build a hash table for tuples
       read hash table for  Ri into memory };
 for  Si do
       { for each tuple in  Si do
             hash the join attributes s(b)
             if match found in  Ri then
             concatenate the two tuples
             place in output relation Q }};
```

이 알고리즘의 경우 일반적으로 병렬 수행을 대상으로 하여 만들어진다. 이 경우, $O((n+m)/K)$의 수행시간이 각 K개의 memory bank에서 일어난다. 그리고, 총 $(2\times K)$개의 프로세서가 사용된다. 하지만, 병렬 수행이 아닌 경우에도 사용되어지는 예가 많다.

2.3.2.2 Hybrid hash join method

hybrid hash join method의 경우는 간단히 알고리즘만을 살펴보겠다. 이 알고리즘은 GRACE hash join 알고리즘 유사한 형태로서 적어도 GRACE hash join 알고리즘정도의 성능을 나타내는 것으로 알려져 있다.

```
for each tuple r do
 { hash the join attributes r(a);
 if hash value lies range for partition R1
```

```
 then
        insert entry in hash table
        else place tuple in appropriate buffer R1 };
flush all buffers except R1 to disk;

for each tuple s do
 { hash the join attributes s(b);
 if hash value lies in range for partition S1
 then
        initiate join process with tuples in R1
 else place tuple in appropriate buffer Si};
flush all buffers to disk;

for i=2,...,n do
 { read tuples in Ri into memory
 build a hash table for Ri};
 for each Si do
        {for each tuple in Si do
               hash the join attributes s(b);
               if match found in Ri then
                      concatenate the two tuples
                      place in output relation Q };
```

제 3장. 알고리즘의 구현

관계 데이타베이스의 그룹 연산과 Count 연산을 위한 중첩루프(nested-loops) 알고리즘, 정렬-합병 알고리즘, 해쉬 알고리즘을 실제로 구현하고 실험하기 위하여 c++을 사용하여 5개의 프로그램을 실제 제작하였다. 본 논문에서 구현한 내용은 다음과 같은 SQL문에 대해 내부적으로 수행하는 알고리즘이다.

```
SELECT COUNT(*)
    FROM R
    GROUP BY R.a
```

제작환경은 Unix기반 하에서 g++ compiler를 사용하였다. 제작한 프로그램은 다음과 같다.

- ㅇ **genfile** : 임의의 파일을 생성하는 프로그램으로서, $m \times n$의 임의의 table file을 random하게 생성한다. 각 field의 domain은 0~99999이다. 각 file은 헤더와 함께 4byte의 integer로 저장되어 있다.
- ㅇ **viewfile** : 만들어진 file을 화면에 출력해주는 프로그램이다.
- ㅇ **g_loop** : 중첩루프(nested-loops) 알고리즘을 이용하여 주어진 file에 대해 그룹 연산과 Count를 한다.
- ㅇ **g_sort** : 정렬-합병 알고리즘을 이용하여 주어진 file에 대해 주어진 memory buffer를 가지고 그룹 연산과 Count를 한다.
- ㅇ **g_hash** : 해쉬 알고리즘을 이용하여 주어진 file에 대해 주어진 memory buffer와 주어진 hash bucket개수를 가지고 그룹 연산과 Count를 한다.

g_loop, g_sort, g_hash 프로그램은 각각 수행시간을 체크하고 이를 log file에 기록한다.

3.1 중첩루프(Nested Loops) 알고리즘

loop을 돌며 한 튜플에 대해 튜플 전체를 스캔하며 같은 개수를 헤아린다. 실제의 구현 역시 대단히 간단하다. 프로그램의 핵심인 튜플의 개수를 세는 알고리즘은 다음과 같다. 한 튜플을 읽어 이보다 앞에 같은 attribute값을 가진 튜플이 있었으면 이는 이미 센 튜플이므로 다음 튜플을 읽고, 그렇지 않으면 개수를 세야 하므로 파일의 나머지 부분에서 같은 attribute값을 가진 튜플의 수를 제어 그 개수를 결과

파일에 기록한다. 이 알고리즘의 경우 Tuple의 개수가 n이라 하면, 이 알고리즘은 최악의 경우 n^2번의 Disk Read를 하게 되고, n번의 Disk Write를 하게 된다.

3.2 정렬-합병(Sort-merge) 알고리즘

정렬-합병(Sort-merge) 알고리즘의 경우 sort방법에 따라 알고리즘의 구현내용이 크게 달라진다. 따라서, 소트 방법을 무엇을 사용하느냐를 결정하는 것이 중요하다. 우선 메인 메모리의 크기보다 파일의 크기가 클 것이기 때문에 외부 합병 정렬을 사용하였다. 메인 메모리를 가장 효율적으로 사용하며 정렬을 할 수 있는 방법이기 때문에 외부 합병 정렬을 선택하였다. 외부 합병 정렬 방법도 여러 가지가 있을 수 있다. Run의 개수와 Disk I/O를 줄이기 위한 여러 가지 개선된 알고리즘이 소개되어 있다. Run의 크기를 서로 다르게 하고, 각 런의 합병 순서를 정하기 위하여 huffman code를 이용하거나, Double Buffering기법을 이용하면 성능을 최적화할 수 있다. 본 논문에서는 이 중에서 가장 간단한 2-way merge 소트를 사용하였다. 그리고 각 런의 내부정렬은 Quick sort를 이용하였다. Quick sort를 사용함으로써 메인 메모리 내부에서는 $O(n\log n)$의 수행 속도를 기대할 수 있다. 외부에서도 2-way merge sort이므로 같은 complexity를 가지지만, n-way merge나 Double buffering 기법에 비하여는 Disk I/O의 회수나 런의 합병 회수가 커지므로 성능에서는 약간 뒤떨어진다.

구현 내용의 수행 알고리즘을 살펴보면 아래와 같다.

① 파일을 순차적으로 읽어가며 선택된 필드를 buffer의 크기만큼 런을 만든다.
② 런을 Quick-sort로 내부 정렬하여 파일에 저장한다.
③ 모든 런이 만들어 질 때까지 ①을 반복한다.
④ 각 런을 2개씩 합병 정렬하여 새로운 런을 생성 파일에 저장한다.
⑤ 모든 런이 합병될 때까지 ④를 반복한다.
⑥ 소트가 된 최종 런을 순차적으로 읽으며 각 값의 개수를 세고 저장한다.

그림 1은 위의 내용을 그림으로 설명한 내용이다.

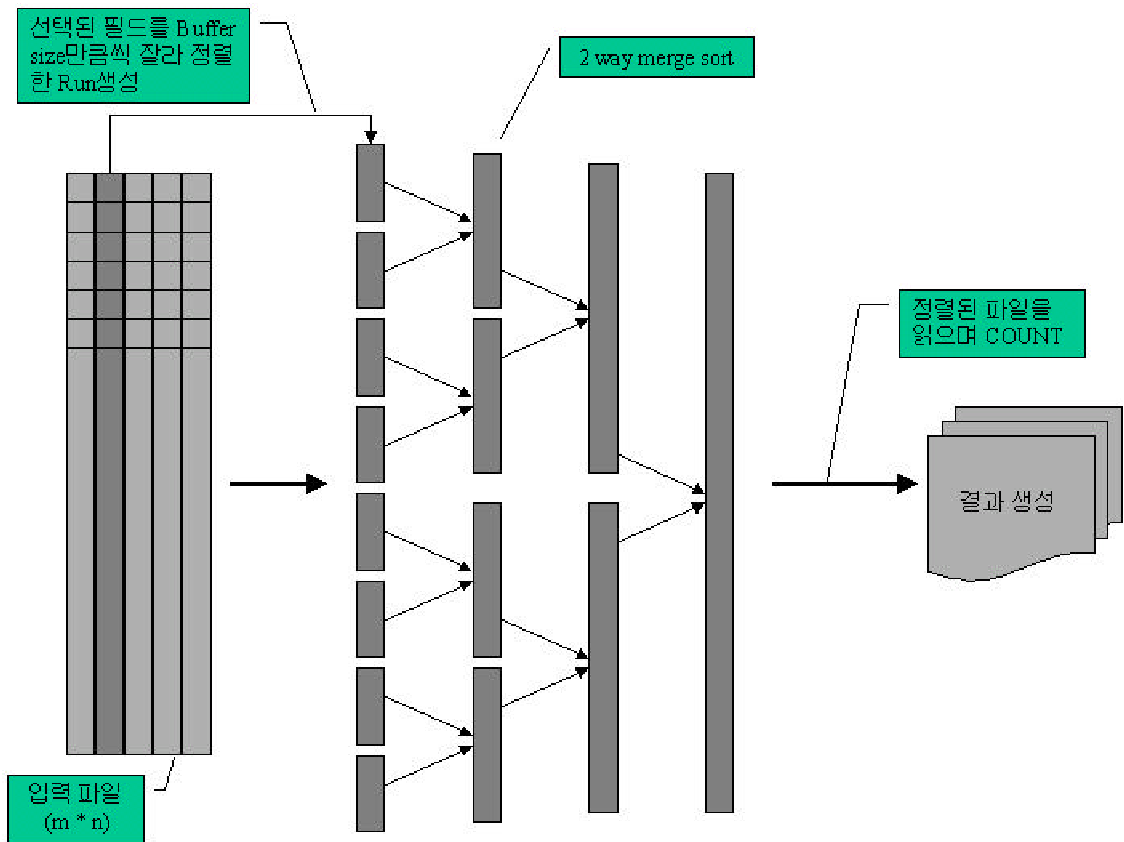


그림 1 정렬-합병(Sort-merge) 알고리즘의 수행과정

3.2.1 내부 정렬

위의 구현 내용 중 내부 정렬은 일반적인 Quick Sort를 사용하였다. Quick-Sort의 구현은 참고문헌[6]을 참조하였다. 다음은 Quick-Sort의 실제 구현 Code이다.

```
void QuickSort(int *buf,int left,int right)
{
 if (left<right) {
        int i=left,j=right+1,pivot=*(buf+left);

        do {
               do i++; while((*(buf+i)<pivot)&&(i<=right));
               do j--;while((*(buf+j)>pivot)&&(j>=left));
               if (i<j) InterChange(buf,i,j);
        }while (i<j);
```

```
            if (left!=j) InterChange(buf,left,j);

            QuickSort(buf,left,j-1);
            QuickSort(buf,j+1,right);
    }
  }
```

3.2.2 외부정렬

외부 정렬의 경우 참고문헌[7]의 파일의 합병 정렬법을 응용하였다. 메모리 Buffer를 최대한 이용하고 Disk I/O의 회수를 줄이기 위하여 메모리의 Buffer를 두 개의 input buffer와 하나의 output buffer로 나누어 사용하였다. 전체 buffer의 1/2는 output buffer로 나머지 1/2을 두 개의 input buffer로 사용하여 한번에 buffer크기 만큼씩 Disk I/O를 수행하여 전체 disk I/O의 횟수를 줄였다.

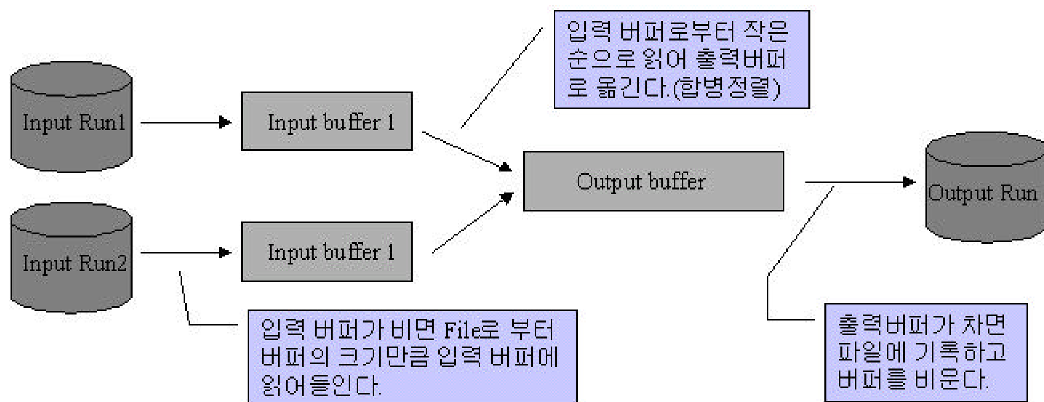


그림 2 합병정렬 방법

그림 2는 이 방법을 그림으로 설명한 내용이다.

3.3 해쉬 알고리즘

해쉬 알고리즘의 경우 파일의 크기가 매우 크기 때문에 hash bucket의 overflow 처리가 주요한 구현의 핵심이 된다. hash bucket의 overflow를 해결하기 위한 방법도 여러 가지가 소개 되어있다. 앞에서 설명했듯이 GRACE 해쉬 알고리즘이나, Hybrid 해쉬 알고리즘처럼 hash bucket을 동적으로 할당하는 방법도 있고, overflow area를 별도로 두거나 extendable hashing기법을 사용하는 경우도 많다. 하지만, 실제 구현에 있어서는 구현의 간단함 때문에 해쉬 알고리즘을 응용하여 구현하였다.

memory에 hash table을 두고 각 bucket의 크기를 고정시킨다. 그리고 파일로부터 튜플을 읽어들여 hash function에 의해 만들어진 hash value를 얻는다. 이 hash value에 따라 해당 bucket에 읽어들인 값을 저장한다. 이 때, bucket이 가득 차면 그 bucket을 해당되는 hash file에 저장하고 bucket을 비운다. 이렇게 하면, 최종적으로 각 hash file에 같은 hash value를 가지는 튜플들이 모이게 된다. 그림 3은 이러한 과정을 그림으로 표현한 내용이다.

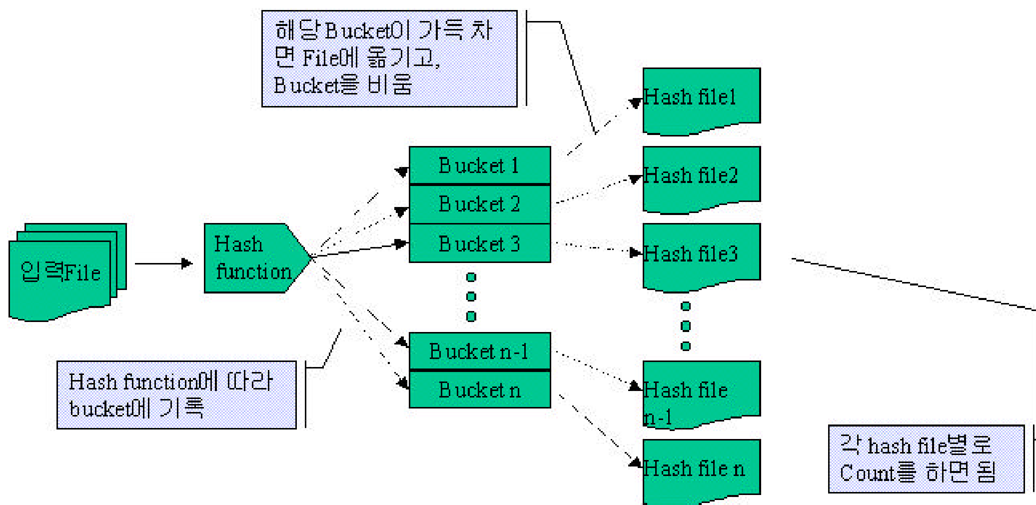


그림 3 해쉬 알고리즘수행 과정

각 hash file은 메모리에 한번에 읽혀질 수 있으므로, 메모리 내에서 nested loops와 유사한 방법으로 count를 하면 된다. 물론 hash bucket에 overflow가 났을 때, 이를 해결하는 더 우수한 방법들이 있지만, 본 논문에서는 이상과 같은 방법을 이용한다. 따라서, 최적화된 알고리즘보다는 Disk I/O의 횟수가 늘어난다. 그리고, bucket의 수만큼 나중에 count를 위해 file을 읽어야 하므로 여기서 생기는 overhead도 존재한다.

제 4장. 실험결과 및 분석

실험은 3장에서 구현된 프로그램에 의하여 다음과 같은 환경에서 수행하였다.

CPU	Pentium Pro 200
RAM	64M
OS	Linux

표 1 시스템 사양

실험대상 파일은 다음과 같다.

튜플의 개수	1000,5000,10000,20000,30000,50000,100000,500000,10000000
필드의 개수	5
필드의 도메인	0~99999인 정수

표 2 실험 대상 파일

각 실험은 3회 실행하여 수행시간과 Disk I/O의 횟수를 측정하였다. 아래의 각 자료는 이 실험 값의 평균값이다. 각 시간 값의 단위는 sec이다.

4.1 실험 결과

Tuple	loop	sort(1000)	sort(5000)	hash(1000)	hash(5000)
100	0.01	0.00	0.02	0.05	0.04
1000	1.73	0.03	0.01	0.06	0.08
5000	39.57	0.29	0.15	0.13	0.12
10000	155.40	0.60	0.36		0.37
20000	620.30	1.22	0.59		0.47
30000	1389.21	2.09	0.91		0.84
50000		2.61	1.23		1.25
100000		4.23	1.49		
500000		51.56	14.85		
1000000		247.09	116.56		
5000000					

표 3 튜플의 수에 따른 실험 값 1

아래는 튜플의 수에 따른 중첩루프(nested-loops) 알고리즘과 메모리의 크기를 1000과 5000으로 주고 측정한 정렬-합병(Sort-merge) 알고리즘과 해쉬 알고리즘의 수행시간이다.

중첩루프(nested-loops) 알고리즘의 경우는 튜플의 개수가 50000을 넘었을 경우부터는 측정하지 않았다. 이는 수행시간이 너무 증가하여 다른 알고리즘과의 비교가 무의미했기 때문이다. 해쉬 알고리즘의 경우도 튜플의 개수가 증가했을 때의 값이 메모리의 크기가 상대적으로 작아 해쉬 테이블의 버킷의 크기가 너무 작아 sort-merge와의 비교가 무의미했기 때문에 측정하지 않았다.

Tuple	hash(10000)	sort(10000)	sort(20000)	hash(20000)	sort(30000)	hash(30000)
10000	0.32	0.21				
20000	0.46	0.52				
30000	0.72	0.90				
50000	0.93	0.88	0.77	1.03	0.74	0.96
100000	1.92	1.07	0.97	2.13	0.88	1.73
500000	18.13	14.92	10.02	16.95	10.52	16.83
1000000	693.39	98.27	58.21	243.08	43.30	172.24
5000000					290.84	1185.28

표 4 튜플 수에 따른 실험값 2

위의 표는 튜플의 수가 10000,20000,30000개인 경우의 실험값이다.

Tuple	sort(50000)	hash(50000)	sort(100000)	hash(100000)
50000	0.59	0.81	0.52	6.11
100000	1.62	2.10	0.75	9.49
500000	8.18	16.30	6.76	28.31
1000000	39.90	118.83	30.61	80.40
5000000	260.52	935.40	237.09	707.90

표 5 튜플 수에 따른 실험값3

위의 표는 메모리의 크기가 50000,100000byte일 경우의 정렬-합병(Sort-merge) 알고리즘과 해쉬 알고리즘의 실험결과이다. 튜플의 수가 5000000개일 때는 성능이 갑자기 나빠졌는데, 이는 5000000개일 경우는 파일의 크기가 100Mbyte가 넘기 때문에, run이 천개가 넘게 생기기 때문에 Disk I/O의 횟수가 늘어날 뿐 아니라, Disk I/O시의 수행 성능이 나빠졌기 때문이다.

Tuple	sort(500000)	hash(500000)	sort(1000000)	hash(1000000)
50000	0.33	10.42	0.39	13.47
100000	0.63	10.52	0.69	16.11
500000	4.15	17.62	3.53	21.43
1000000	16.98	26.32	10.22	27.84
5000000	186.73	497.84	172.19	438.77

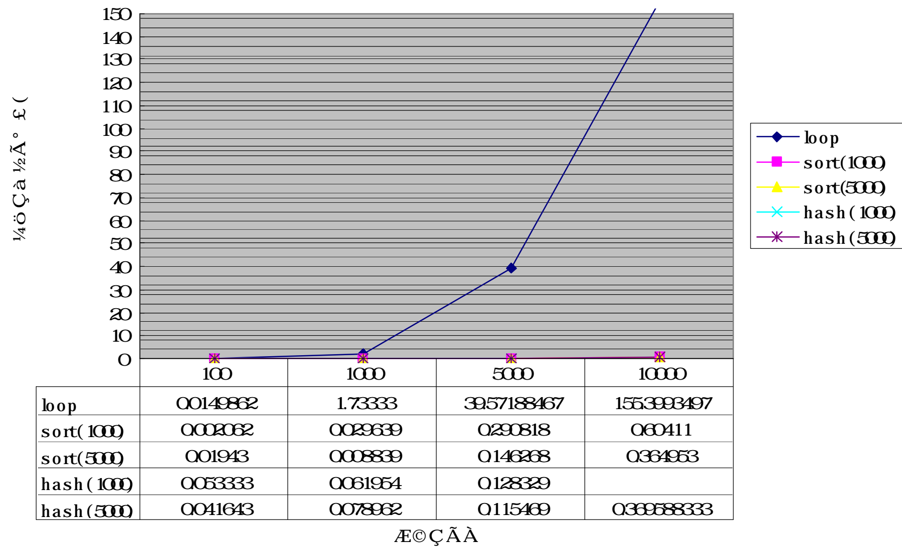
표 6 튜플 수에 대한 실험값4

위의 표는 메모리의 크기가 약 500Kbyte일 때와 약 1Mbyte일 경우의 정렬-합병(Sort-merge) 알고리즘과 해쉬 알고리즘의 수행시간이다. 파일의 크기가 20Mbyte인 경우(튜플의 수가 1000000개)까지는 수행성능이 아주 우수하다. 하지만, 파일의 크기가 100Mbyte가 넘을 때는 역시 수행시간이 현저히 나빠졌다.

4.2 실험 결과 분석

4.2.1 튜플의 수에 따른 중첩루프 알고리즘의 수행 성능

아래 그림에서 볼 수 있듯이 튜플의 수가 커짐에 따라 다른 알고리즘의 수행 속도는 크게 증가하지 않았지만, 중첩루프(nested-loops) 알고리즘의 경우는 튜플 수에 대해 급격히 증가하였다. 이는 중첩루프(nested-loops) 알고리즘이 튜플의 수 n에 대하여 $O(n^2)$의 수행시간을 가지기 때문이다.



	100	1000	5000	10000
loop	0.0149862	1.73333	39.57188467	155.3993497
sort(1000)	0.002062	0.029639	0.290818	0.60411
sort(5000)	0.01943	0.008839	0.146268	0.364953
hash(1000)	0.053333	0.061954	0.128329	
hash(5000)	0.041643	0.078962	0.115469	0.369588333

그림 4 튜플의 수가 100개에서 10000개까지의 각 알고리즘의 수행성능

이러한 내용은 중첩루프(nested-loops) 알고리즘의 Disk I/O횟수에서도 잘 알 수 있다.

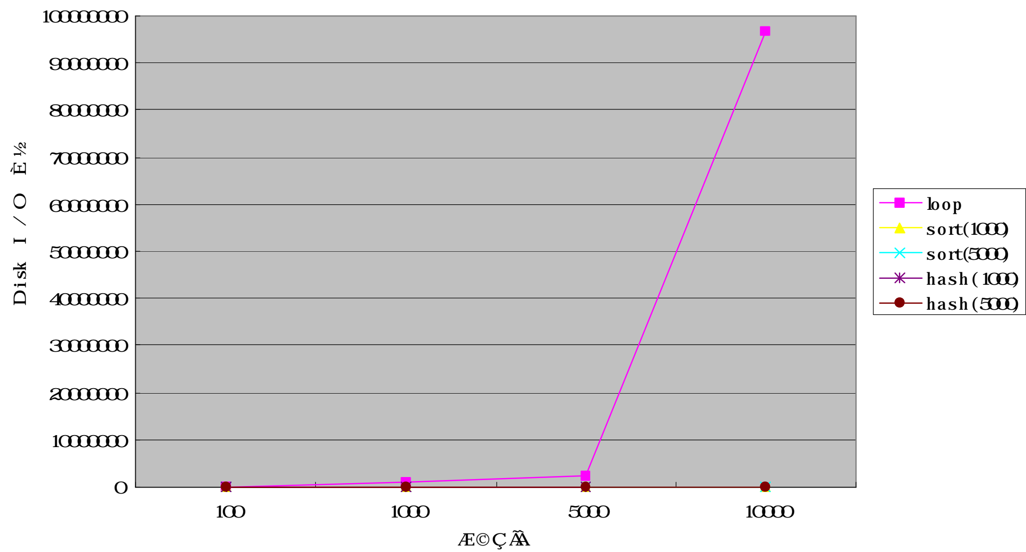


그림 5 튜플의 개수에 따른 Disk I/O의 횟수

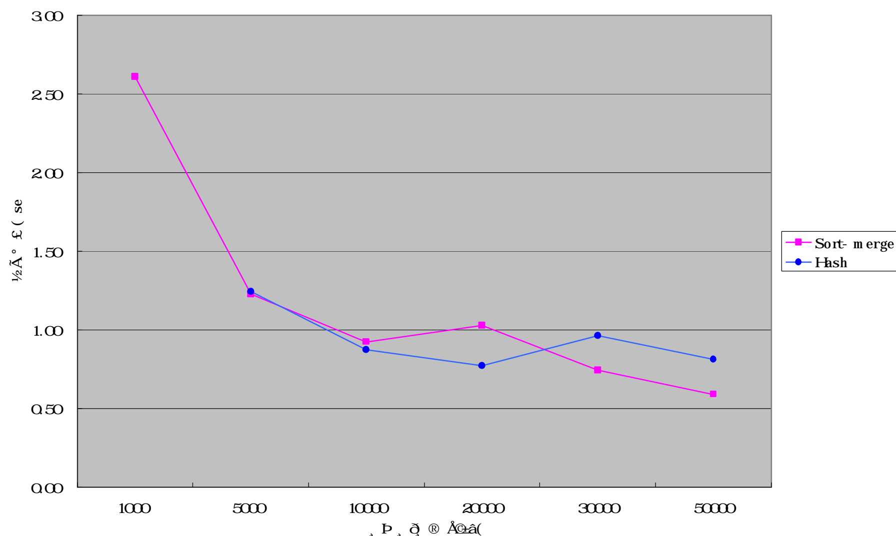
이 그림에서 알 수 있듯이 중첩루프(nested-loops) 알고리즘은 튜플의 개수에 따라 급격히 Disk I/O횟수가 증가한다. Disk I/O의 경우 수행 시간에 가장 큰 영향을 미침을 알 수 있다.

따라서, 중첩루프(nested-loops) 알고리즘은 튜플의 수가 아주 작을 때는 효용의 가치가 있으나, 튜플의 수가 많을 때는 일반적인 소프트웨어 적인 방법으로는 효용 가치가 없다.

4.2.2 메모리의 크기에 따른 수행 시간 분석

중첩루프(nested-loops) 알고리즘의 경우는 메모리를 사용하지 않기 때문에 정렬-합병(Sort-merge) 알고리즘과 해쉬 알고리즘에 대하여 메모리의 크기를 바꿔가며 실험한 값을 분석해 보았다.

다음은 튜플의 수가 20000개일 때의 정렬-합병(Sort-merge) 알고리즘과 해쉬 알고리즘의 수행 시간이다.



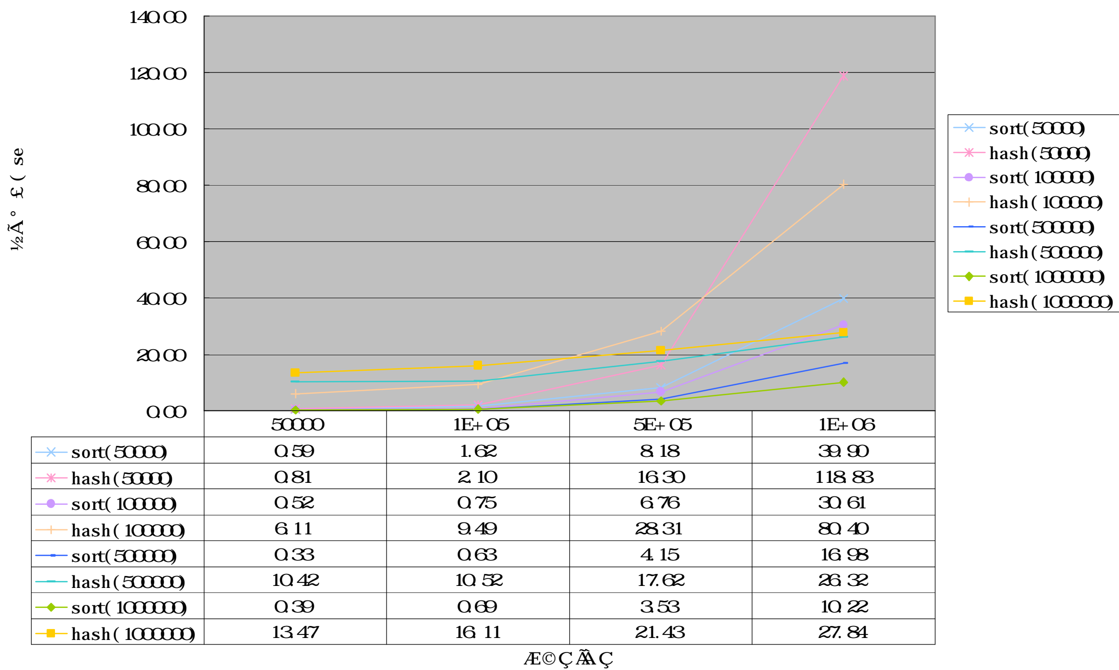
이 그래프를 보면 메모리의 크기가 커짐에 따라 수행시간이 현저히 줄어드는 것을 알 수 있다.

Sort-merge의 경우에는 메모리의 크기가 커질수록 런의 개수가 적어지므로 합병에 필요한 메모리 I/O가 줄어들기 때문이다. 이러한 현상은 튜플의 수가 더 많을 때에도 동일하게 일어난다. 한 런의 크기가 커지므로 내부정렬 시간은 커지지만, 이는 메모리 내부에서 일어나는 시간이므로, Disk I/O에 걸리는 시간보다 현저히 작은 시간의 증가이므로 전체적으로 성능은 월등히 좋아진다.

hash의 경우도 메모리가 커짐에 따라 성능이 좋아진다. hash의 경우 메모리가 커

지면 hash의 overflow가 상대적으로 줄어들게 된다. 따라서, hash table에서의 Disk I/O의 횟수가 줄어든다. 따라서 memory의 크기가 20000byte정도일 때는 정렬-합병(Sort-merge) 알고리즘보다 성능이 더 좋게 나타난다. 하지만 memory가 50000byte보다 커졌을 때는 성능이 약간 둔화되었는데, 이는 이미 메모리에 overflow가 거의 없기 때문에 개수를 셀 때의 Disk I/O는 거의 같으므로, 성능이 거의 유사하게 나타난다.

4.2.3 튜플의 수에 따른 정렬-합병 알고리즘과 해쉬 알고리즘의 성능



	50000	1E+05	5E+05	1E+06
sort(50000)	0.59	1.62	8.18	39.90
hash(50000)	0.81	2.10	16.30	118.83
sort(100000)	0.52	0.75	6.76	30.61
hash(100000)	6.11	9.49	28.31	80.40
sort(500000)	0.33	0.63	4.15	16.98
hash(500000)	10.42	10.52	17.62	26.32
sort(1000000)	0.39	0.69	3.53	10.22
hash(1000000)	13.47	16.11	21.43	27.84

그림 7 튜플의 개수에 따른 수행시간

위 의 그림에서 알 수 있듯이 튜플의 개수에 따라 수행 시간이 점차적으로 증가해 간다. 하지만, 튜플의 수가 50000에서 1000000개로 20배 증가하더라도 대부분의 경우 수행시간은 그리 큰 증가를 보이는 것은 아님을 알 수 있다.

50000byte의 메모리를 가진 해쉬 알고리즘의 경우에만 튜플의 수가 100만개일 때 수행시간이 급격히 증가하는데, 이는 튜플의 수에 비해 버킷 하나의 크기가 너무 작기 때문으로 보인다.

위의 결과에서도 알 수 있듯이 메인 메모리가 충분하다면 sort-merge나 해쉬 알고리즘은 그룹연산을 처리하는데 짧은 응답시간을 보장해 줄 수 있으므로, 아주 유

용하게 사용될 수 있음을 알 수 있다.

4.3 각 알고리즘에 대한 평가

4.3.1 중첩루프(nested-loops) 알고리즘

중첩루프(nested-loops) 알고리즘의 경우 수행시간이 튜플의 개수에 따라 아주 큰 변화를 나타내었다. 튜플의 수의 제곱에 비례하여 시간이 증가하므로 튜플의 수가 클 때는 효용가치가 없는 것으로 판명되었다. 하지만, 메인 메모리에 별도의 버퍼가 필요 없고, 알고리즘의 구현이 간단하므로 튜플의 크기가 많지 않은 아주 간단한 응용이나, 메인 메모리의 크기에 제한을 많이 받는 경우에는 유용하게 쓰일 수 있을 것으로 보인다. 특히 구현이 간단하므로 하드웨어적으로 구현하기도 쉬울 것이므로 간단한 하드웨어 그룹연산 처리기는 이 알고리즘을 이용하여서도 구현할 수 있을 것으로 판단되고, 이러한 방법을 통하여 속도의 문제를 보완한다면 사용이 가능한 알고리즘이다.

4.3.2 정렬-합병(sort-merge) 알고리즘

정렬-합병(sort-merge) 알고리즘의 경우 가장 우수한 성능을 보였다. 이는 메모리 버퍼를 효율적으로 이용함으로써 Disk의 I/O횟수를 최소화 할 수 있었기 때문이다. 특히 내부정렬과 외부정렬 모두 $O(n\log n)$의 수행 성능을 가지는 알고리즘을 사용하였으므로, 수행시의 오버헤드를 최소화하였다. 정렬-합병 알고리즘의 경우 메모리의 크기만 어느 정도 수준이 넘으면 튜플의 크기가 커지더라도 수행 시간의 증가가 그리 크지 않았다. 이는 일정 한도내의 응답시간을 보장할 수 있다는 의미이므로 대규모의 응용에도 적합한 알고리즘이라 할 수 있다.

본 논문에서는 메모리의 크기에 따라 런의 크기를 고정시키고, 이원 합병만을 하였으나, 차후 입력 파일의 특성에 따라 런의 크기를 유동적으로 조정할 수 있고, n원 합병을 가능하게 하며, 합병시 버퍼를 더 효율적으로 사용한다면, 더욱 효율을 극대화 할 수 있을 것이다.

정렬-합병(sort-merge) 알고리즘의 경우는 각각의 런들에 대하여 병렬적으로 모든 작업이 수행될 수 있으므로, 병렬적인 방법으로 구현한다면 보다 좋은 수행성능을 기대할 수 있을 것이다.

4.3.3 해쉬 알고리즘

해쉬 알고리즘의 경우 일반적으로는 아주 우수한 성능을 보였다. 하지만, 경우에 따라서는 정렬-합병(sort-merge) 알고리즘에 비해 성능이 많이 나쁜 경우가 있었다. 이는 해쉬 알고리즘의 구현 시 가장 간단한 알고리즘에 의해 구현하였으므로, 버킷의 크기나 개수가 입력 파일의 크기와 메모리의 크기에 적합하지 않아 필요 이상의 오버헤드가 발생하였기 때문이다. 실험결과 hash의 경우 메모리의 크기가 일정하더라도, 버킷의 개수와 한 버킷의 크기에 따라 수행 성능은 다르게 나타났다. 이로 미루어 hash의 경우 메모리의 크기 못지 않게, 버킷의 크기나 개수도 중요한 요인임을 알 수 있다.

본 논문에서는 평균적으로 정렬-합병(sort-merge) 알고리즘에 비해 해쉬 알고리즘의 성능이 떨어지는 것으로 나타났다. 하지만, 일반적으로 다른 연구를 살펴보면 hash의 성능이 가장 뛰어난 것으로 알려져 있다. 이는 본 논문의 경우 해쉬 알고리즘만을 대상으로 하였지만, 다른 논문의 경우 GRACE 해쉬 알고리즘이나 Hybrid 해쉬 알고리즘과 같이 보다 향상된 알고리즘을 사용했기 때문이다. 이러한 향상된 알고리즘들은 해쉬 버킷을 동적으로 관리함으로써 보다 메모리를 효율적으로 사용하여 Disk I/O의 횟수를 최소화시키고 있다.

해쉬 알고리즘의 경우 정렬-합병(sort-merge) 알고리즘과 마찬가지로 수행성능이 튜플의 수에 대해 급격히 달라지지 않고 일반적으로 우수한 성능을 보이므로 여러 응용 프로그램에서 응용될 수 있을 것이다. 그리고 이 알고리즘 역시 병렬적으로 수행될 수 있으므로 병렬적으로 구현되어 진다면 보다 우수한 성능을 나타낼 것이다.

제 5장. 결론 및 향후의 연구 과제

5.1 결론

본 논문에서는 관계 데이타베이스 시스템의 연산중 그룹연산의 내부 구현 알고리즘 세가지(중첩루프(nested-loops) 알고리즘, 정렬-합병(sort-merge) 알고리즘, 해쉬 알고리즘)를 구현하고, 메모리의 크기와 튜플의 수에 대한 수행 시간을 측정함으로써 세가지 알고리즘을 비교 분석하였다. 이를 통하여 메인 메모리의 크기가 점점 커지고 마이크로 프로세서의 성능이 좋아지는 등 전반적인 데이타베이스 시스템의 성능 향상에 따른 세가지 알고리즘의 효용을 검증해 보았다. 그 결과 중첩루프(nested-loops) 알고리즘의 경우 튜플의 수가 작거나 메모리의 제약이 큰 경우 사용할 수 있음을 알 수 있었고, 정렬-합병(sort-merge) 알고리즘과 해쉬 알고리즘의 경우 메모리의 크기에 따라서 튜플의 수가 많더라도 효과적인 수행이 가능함을 알 수 있었다. 따라서, 정렬-합병(sort-merge) 알고리즘과 해쉬 알고리즘의 경우 현대의 고성능, 대용량의 시스템에 적합한 알고리즘임을 알 수 있었다.

5.2 향후 연구과제

본 논문에서 구현된 알고리즘은 최적화하기 위해서는 보완할 점이 많은 알고리즘이다. 정렬-합병(sort-merge) 알고리즘의 경우 double buffering기법이나 n-way 합병기법을 도입 더욱 향상된 성능을 분석할 필요가 있다. 단순 해쉬 알고리즘의 경우 역시 GRACE 해쉬 알고리즘이나 Hybrid 해쉬 알고리즘과 같은 보다 우수한 성능을 보일 수 있는 알고리즘으로 개선 분석한다면 더 좋은 결과를 얻을 수 있을 것이다. 그리고, 정렬-합병기법이나 해쉬 기법은 병렬적으로 구현될 수 있는 알고리즘이므로 병렬적으로 구현하여 향후 다중 프로세서 환경에서의 성능분석도 필요하다.

참고문헌

[1] Blasgen, M.W. and K.P. Eswaran, "Storage and Access in Relational Database", IBM Systems Journal, Vol. 16, No. 4, 1977

[2] Kitsugawa, M. et al, "Application of Hash to Database Machine and its Architecture", New Generation Computing, No.1, pp 62-74, 1983

[3] Dewitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., and Wood, D. ,"Implementation techniques for main memory database systems", Proceedings of SIGMOD, pp 1-8, 1984

[4] Priti Mishra and Margaret H.Eigh, "Join Processing in Relational Database", ACM Computing Surveys, Vol.24, No. 1, March 1992,

[5] Su, S. Y. W. "Database Computers: principles,Architectures,and Techniques", McGrawHill, New York, 1988

[6] 이석호, "C++ 자료구조론", 교보문고, 1996

[7] 이석호, "화일 처리론", 정익사, 1985