Assignment2 Wiki

2017030055윤상현

이번 project에선 3가지 join situation이 주어지고, 각 케이스에 대해 적절한 join 알고리즘을 구현하는 project입니다. 각 Case는 다음과 같습니다.

Case 1) name_age, name_salary 두개의 table을 join 하되, name attribute가 sort되어있다.

Case 2) name_age, name_salary 두개의 table을 join 하되, age, salary attribute가 sort되어있다.

Case 3) name_grade1, name_grade2, name_number 3개의 table에서, grade1보다 grade2에서 성적이 향상된 학생을 찾고, name_number table을 통해 학생의 이름과 번호를 찾아라.

각 Case에 대하여 제가 생각한 join 알고리즘과, 그 방식, 이유에 대하여 설명하겠습니다.

Case1

Case1의 경우는 name attribute를 기준으로 natural join합니다. 즉 name attribute 가 같을 때 Join 하는 equi join이고, 이 경우 sort가 되있다면 merge join, 그렇지 않다면 hash join 이 nested join에 비해 유리합니다. 이 경우, join할 attribute가 sort 되어있기에 merge join이 적합하다 생각하여 merge join을 통해 구현했습니다. 아래는 코드의 일부입니다.

```
    int br=0,bs=0;
    int i=0;
    temp0=name_age();
    temp1=name_salary();
    block[1].open("./name_salary/0.csv");
    getline(block[1],buffer[1]);
    temp1.set_name_salary(buffer[1]);
    //here, eval equality?
    int chk=1;
while(br!=1000&&bs!=1000){// ~10th block
    bool done1=false;

    while(br!=1000&&!done1){

       if(chk==1){//if need to open new, next block
          block[i].close();
     // cout<<"outer block change!"<<endl;
      string line;
      char num='0';
      string f="./name_age/";
      string b=".csv";
      string ss;
      ss+=f;
      ss+=to_string(br);
      ss+=b;
      block[i].open(ss);
      chk=0;
      }
```

Br, bs 변수는 각각 outer table인 name_age, inner table인 name_salary table의 파일 index 입니다.

name_age, name_salary 폴더에는 각각 1000개의 csv 파일이 있고, 이 csv 파일들을 통해 1개의 table을 구성하기에 search 할 때 현재 내가 name_age table의 몇 번째 csv 파일을 탐색하고 있는지 저장하는 변수입니다. 가장 바깥의 while의 종료 조건은 두 table 모두 1000이 넘게 되는 시점이며, 이는 각 테이블 내의 모든 csv 파일을 탐색했을 때를 나타냅니다.

그리고 if(chk==1) 로 감싸진 부분은 해당하는 outer table의 br 번째 csv 파일을 로드하는 코드로,

만약 한 개의 i.csv 파일을 모두 탐색했다면, (i+1).csv 파일을 탐색해야 하기에, path 명으로 open 할 때 path 명에서 파일 이름을 br로 설정한 뒤 open 하는 것으로 처리했습니다.

```
if( block[i].is_open() ){

    if(!block[i].eof()) {
    getline(block[i],buffer[0]);
    temp0.set_name_age(buffer[0]);
     if(block[i].eof()){ //if next tuple does not exist, previously move to  next block
          br++;
          cout<<"outer block end, go to next!"<<endl;
          chk=1;
            continue;
     }
     //cout<<"na: "<<na.name<<"ns: "<<ns.name<<endl;
      if(temp0.name.compare(temp1.name)==0){//nat join two tuple
                string test2 =make_tuple(temp0.name,temp0.age,temp1.salary);//add to join table and continue
            //  cout<<"find1: " <<test2;
                output<<test2;
                cnt++;
      }
      else if(temp1.name.compare(temp0.name)==-1  ){
                  done1=true;

      }
      else{
          done1=true;
      }
    }
```

위 코드는 outer table의 name attribute > inner table의 name attribute 가 될 때까지

getline() 함수를 통해 Outer table의 탐색 중인 csv 파일을 index를 1줄씩 read합니다. 이 과정에서 만약 name filed 가 같은 attribute를 발견한다면, 같은 attribute를 가진 두 tuple을 합친 뒤, output 파일에 작성해줍니다.

```
    bool done=false;
    int j=1;
    int chk2=0;
    while(bs!=1000&&!done){
        if(chk2==1){
            block[1].close();
        //  cout<<"innter block change!"<<endl;
            string ff="./name_salary/";
            string bb=".csv";
            string ssss;
            ssss+=ff;
            ssss+=to_string(bs);
            ssss+=bb;
            block[j].open(ssss);
            chk2=0;
        }
        if( block[j].is_open() ){
              if(!block[j].eof()){
                 getline(block[j],buffer[1]);

//               cout<<"inner: "<<buffer[1]<<endl;
                 if(block[j].eof()){//if next tuple does not exist,
//                   cout<<"eof: "<<buffer[1]<<endl;
                     bs++; //if one block done, next block!
                     chk2=1;
                     continue;
                    // ps=0;//reset!
                 }
```

이후부턴 inner table에서 탐색을 진행합니다. Inner table 역시 outer table과 마찬가지로

여러 csv 파일로 구성되어 있기에, 현재 탐색할 csv 파일을 open하고, 만약 해당 csv 파일의 끝까지 탐색했다면 다음 csv 파일을 open 합니다.

```
while(bs!=1000&&!done){
    if(chk2==1){
        block[1].close();
    //  cout<<"innter block change!"<<endl;
        string ff="./name_salary/";
        string bb=".csv";
        string ssss;
        ssss+=ff;
        ssss+=to_string(bs);
        ssss+=bb;
        block[j].open(ssss);
        chk2=0;
    }
    if( block[j].is_open() ){
          if(!block[j].eof()){
             getline(block[j],buffer[1]);
             if(block[j].eof()){//if next tuple does not exist,
                   bs++; //if one block done, next block!
                   chk2=1;
                   continue;
              }
            temp1.set_name_salary(buffer[1]);
             if(temp0.name.compare(temp1.name)==0){//nat join two tuple
              string test1=make_tuple(temp0.name,temp0.age,temp1.salary);//add to join table and continue
              output<<test1;
              cnt++;
             }
             else if( temp0.name.compare(temp1.name)==-1){
                // cout<<"swip"<<endl;
                 done=true;
             }
```

이후 inner table의 해당 csv 파일에 대해 getline() 함수로 한 줄씩 read 하는데,

Outer table과 마찬가지로 inner table의 name attribute > outer table의 name attribute가 될 때까지 inner table의 tuple 들을 차례로 탐색합니다. 만약 이 과정에서 name attribute 가 같은 tuple을 찾는다면 마찬가지로 output 파일에 write 해줍니다.

이 과정을 통해, merge join은 outer과 inner table의 tuple들을 탐색 할 때, 탐색할 필요가 없는 tuple들을 sorted가 되었다는 전제를 기준으로 배제하며 search 하여 join 성능을 향상하게 됩니다.

Case 2

Case2는 Case1과 다르게 natural join의 기준이 되는 name attribute가 정렬 되어있지 않습니다.

따라서 이는 join시에 sorted 되어 있다는 이점을 사용할 수 없고, merge join을 사용하려면 새롭게 name attribute들을 sort해줘야합니다. 또한 natural join, 즉 특정 attribute가 같은 값을 가지는 조건을 통해 join 하는 equi join 이기에 nested loop join 보단 hash join을 사용하여 성능을 개선할 수 있습니다.

Hash join은 join할 두개의 table 중 크기가 작은 table을 적절한 hash function을 통해 hash table을 만들어 준 뒤, 남은 table의 tuple들을 hash function을 통해 미리 만들어 둔 hash table의 값들과 비교하는 것으로 search time을 개선합니다. 아래는 코드 설명 입니다.

```
int hash(string name) {
    int val = 0;
    for (int i = 0; i < name.length(); i++) {
        val += name[i];
    }
    val = val % 10;
    val += 2;// val은 2~ 11 ( buffer 내부} 가 나온다.
    return val;
}
```

위 코드는 hash function입니다. 함수의 parameter로 tuple의 name attribute를 넘겨주고,

Name은 string type 이기에 여러 개의 char 타입으로 나눕니다. 그리고 해당 char의 아스키 값을 전부 합하여 val 변수를 만들고, 이를 10을 나눈 나머지로 만든 뒤 2를 더합니다.

이는 주어진 block[12] 배열의 크기가 12인 것을 염두하여 구현한 것으로, block[0]와 block[1] 의 공간은 다른 곳에서 사용되기에, 남은 block [2]~ block[11]의 공간을 hash table을 위한 공간으로 구현하였습니다. Hash value는 제 구현에서 들어가 있는 block의 index 정보도 내포하기에

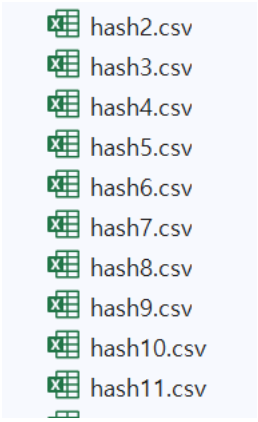
Hash 값이 2~11 사이에 들어가게 만들기 위하여 위와 같은 계산을 추가하여 구현한 것입니다.

```
    for (int i = 2; i < 12; i++) {
        string f = "./hash";
        string b = ".csv";
        string ss;
        ss += f;
        ss += to_string(i);
        ss += b;
        block[i].open(ss,ios_base::out);
        if(block[i].fail()){
          cout << "output1 file opening fail.#n";
      }
//      block[i].close();

    }
```

Hash join을 시작하기에 앞서, 사용할 hash table들을 모두 hashi.csv 파일로 생성합니다.

이후 hash table에 값을 write하거나 read 할 때, 이때 생성한 csv 파일에 정보가 들어가는 것으로 hash table을 구현하였습니다.



Hash 정보를 담은 csv 파일들은 open() 함수를 통해여 현재 디렉토리 내에 위와 같이 생성되게 됩니다.

```
for (int i = 0; i < 1000; i++) {
    string line;
    char num = '0';
    string f = "./name_age/";
    string b = ".csv";
    string ss;
    ss += f;
    ss += to_string(i);
    ss += b;
    block[0].open(ss);
    while (!block[0].eof()) {

            getline(block[0], buffer[0]);
            if (block[0].eof()) continue;
            temp0.set_name_age(buffer[0]);

            int hash_val = hash_func(temp0.name);

            string tuple=make_double(temp0.name, temp0.age);
            cout<<"name: "<<temp0.name<<" hsah: "<<hash_val<<endl;
            if(!block[hash_val].is_open()) cout<<"not open!"<<endl;
            block[hash_val] << tuple;
    //      cout<<tuple<<endl;
    }
    block[0].close();
}
```

모든 전처리를 마친 뒤, hash join을 위해 join할 두 테이블 중 사이즈가 더 작은 table을

Hash function을 통해 hash table에 추가해야 하지만, sample로 주어진 두 table은 모두

10* 1000= 10000개의 tuple로 대략적으로 유사한 사이즈를 가지고 있고, logical하게 생각했을 때

나이보다는 연봉이 더 높은 값을 가지기에 더 긴 string을 가지고 있다는 전제하에 name_age table을 우선적으로 처리하는 것으로 구현하였습니다.

모든 name_age 테이블 속의 block과 tuple들을 getline()으로 read 한 뒤, hash function을 통해 적절한 hash value를 추출한 뒤, 해당 hash_value를 가진 table에 해당 tuple들을 저장합니다.

여기서 사용한 make_double 함수는 name과 age, 두 개의 string 타입을 (name,age) 튜플 형태로 변환하는 함수로, 코드는 아래와 같습니다.

```
string make_double(string name, string val) {

    return name + ',' + val + '₩n';
}
```

이제 name_age 테이블에 대한 hash table을 모두 hash(i).csv 파일에 write했기에, 이를 토대로 name_salary 테이블을 탐색하며 매칭되는 tuple들을 합친 뒤 output에 작성해줘야 합니다.

```
for (int i = 0; i < 1000; i++) {
    string line;

    string f = "./name_salary/";
    string b = ".csv";
    string ss;
    ss += f;
    ss += to_string(i);
    ss += b;
    block[1].open(ss);
    if(!block[1].is_open()){
        cout<<"open fail!"<<endl;
        continue;}
    while (!block[1].eof()) {
        getline(block[1], buffer[0]);
        if (block[1].eof()) continue;
        //cout<<buffer[0]<<endl;
        temp1.set_name_salary(buffer[0]);
        int hash_val = hash_func(temp1.name);
        string f1 = "./hash";
        string b1 = ".csv";
        string ss1;
        ss1 += f1;
        ss1 += to_string(hash_val);
        ss1 += b1;
//      cout<<ss1<<endl;
        block[hash_val].open(ss1,ios_base::in);
        if(!block[hash_val].is_open()){
    //      cout<<"cant open"<<endl;
            continue;}
```

위 코드에선 name_salary 테이블 내의 모든 csv 파일을 탐색하며 해당 csv 파일의 tuple을 Getline() 메소드로 읽습니다. 이후 hash function을 통하여 해당 tuple의 hash value를 찾은 뒤, 해당 hash value를 가진 csv파일을 열어 탐색합니다.

```
            while (!block[hash_val].eof()) {
                if (block[hash_val].eof()) continue;
                getline(block[hash_val], buffer[0]);
//              cout<<"buf: "<<buffer[0]<<" hash val: "<<hash_val<<endl;
                temp0.set_name_age(buffer[0]);
    //          cout<<temp0.name<<" "<<temp1.name<<endl;
                if(temp0.name.compare(temp1.name)==0){
                    string ans=make_tuple(temp0.name, temp0.age, temp1.salary);
                    output << ans;
//                  cnt++;
                }

            }
            block[hash_val].close();
        }
        block[1].close();

    }
```

이후 해당 csv 파일에서 name attribute 가 같은 값을 발견 한다면, 두 tuple을 합친 뒤, output 파일에 write해줍니다. 기존의 nested join은 outer의 각 tuple들이 Inner의 모든 tuple들과 비교연산을 수행해야하는 것에 반에, 이런 hash function을 활용한다면 같은 hash value를 가지는 범위 안에서만 탐색하면 되기에, search time이 크게 개선됩니다.

Case3

Case 3의 경우 모든 attribute들이 sort 되어있지 않고 random하게 놓여있습니다.

따라서 merge join은 사용할 수 없습니다. 또한 작년 점수 < 올해 점수 와 같이 단순한

Equivalence를 확인하는 조건이 아니기에, hash function 역시 사용할 수 없습니다.

결국 사용하는 join은 nested join으로, 보통 작은 table에서 효율적이지만 이 경우 다른 대안이 없어 nested join을 사용하여 구현하였습니다. 제가 구현한 nested loop join 은 크게 4중 반복문으로 구현하였습니다. 각 반복문은 outer의 block 탐색, outer의 선택된 block 내의 tuple 탐색, inner의 block 탐색 , inner의 선택된 block 내의 tuple 탐색으로 구성되어 있습니다.

```
for (int i = 0; i < 1000; i++) {
        string line;
        string f = "./name_grade1/";
        string b = ".csv";
        string ss;
        ss += f;
        ss += to_string(i);
        ss += b;
        block[0].open(ss);
        while (!block[0].eof()) {
            getline(block[0], buffer[0]);
            if (block[0].eof()) continue;
    //      cout<<"outer: "<<buffer[0]<<endl;
            out_cnt++;
            temp0.set_grade(buffer[0]);
```

우선 가장 바깥의 반복문은 Outer table로 설정한 name_grade2의 모든 block을 순차적으로 탐색하고,

그 안의 while 반복문은 해당 block 내의 모든 tuple을 temp0 객체에 할당합니다. name_age 테이블은 1000개의 csv 파일로 이루어졌기에 i가 1000까지 증가하도록 반복문을 구현하였습니다.

```
for (int j = 0; j < 1000; j++) {
    string ff = "./name_grade2/";
    string bb = ".csv";
    string ssss;
    ssss += ff;
    ssss += to_string(j);
    ssss += bb;
    block[1].open(ssss);
```

그 안의 반복문은 inner table인 name_grade2의 모든 csv 파일을 가져오도록 구현되었습니다.

```
while (!block[1].eof()) {
    getline(block[1], buffer[0]);
    if (block[1].eof()) continue;
//  cout<<"inner: "<<buffer[0]<<endl;
    in_cnt++;
    temp1.set_grade(buffer[0]);

    if (temp0.student_name.compare(temp1.student_name) == 0) {//nat join two tuple
        int cnt = 0;
    //  cout<<"find: "<<temp0.student_name<<" "<<temp1.student_name<<endl;
        if (temp0.korean > temp1.korean) cnt++;
        if (temp0.math > temp1.math) cnt++;
        if (temp0.english > temp1.english) cnt++;
        if (temp0.science > temp1.science) cnt++;
        if (temp0.social > temp1.social) cnt++;
        if (temp0.history > temp1.history)cnt++;
        if (cnt >= 2) {
```

그 안의 반복문은 선택된 inner table의 csv 파일에 대하여 getline() 함수를 통해 모든 tuple들을 읽고, 바깥의 반복문에서 구한 temp0에 들어가 있는 outer table의 tuple과 비교 연산을 수행합니다. 주어진 조건은 작년보다 성적이 향상된 과목이 2개 이상인 사람을 찾는 것이고, 숫자가 더 적은 것이 더 높은 성적입니다. 각 사람은 6개의 과목 attribute가 주어져 있기에 6개의 if문으로 대소를 비교한 뒤 만약 성적이 향상되었다면 cnt 값을 1씩 증가시켜 성적이 향상된 과목의 수를 표기합니다. 이후 성적이 향상된 과목이 2개이상이라면, 조건에 부합하는 학생을 찾았기에 다음 과정을 수행합니다.

```
if (cnt >= 2) {

    for (int k = 0; k <1000; k++) {

        string f3 = "./name_number/";
        string b3 = ".csv";
        string path3;
        path3 += f3;
        path3 += to_string(k);
        path3 += b3;
        block[2].open(path3);
        if (block[2].is_open()) {
            while (!block[2].eof()) {
                getline(block[2], buffer[0]);
                if (block[2].eof()) continue;
                temp2.set_number(buffer[0]);
                if (temp2.student_name.compare(temp1.student_name) == 0) {
                    string improve = make_tuple(temp2.student_name, temp2.student_number);//add to join table and continue
                    output << improve;
                }
            }
        }
        block[2].close();
```

다음 과정은 위에서 찾은 학생의 학생 번호를 찾는 것으로, 이 정보는 name_number 테이블에

(student_name, student_number)의 순서쌍으로 들어있습니다. 이 역시 완전 탐색을 통하여

같은 name 값을 가진 두 tuple들을 합친 뒤, output 파일에 write 합니다.

이와 같은 nested loop join은 완전 탐색으로 동작하여 search time이 길지만, 다른 방식을 사용할 사전 조건이 충족되지 않거나, 작은 table의 join 연산에선 고려해볼 만한 선택지입니다.

Trouble Shooting

File read, write 방식이 생소하여 어려움이 있었습니다. 또한 fstream 타입의 경우, read와 write 모두 수행가능한 type인데, ios_base::in,out 과 같이 명시를 해주지 않으면 올바르게 동작을 하지 않아 이를 찾는 데 어려움이 있었습니다. 하지만 한번 해결한 뒤 부턴 read write 방식이 익숙해져 어려움이 사라졌습니다. 또한 hash function에서 hash function을 만드는 것에서 어려움이 발생했습니다. 최대한 주어진 10칸을 균등하게 grouping 할 수 있는 hash function을 고안하는데 어려움이 있었지만, 주어진 name attribute를 최대한 활용하는 방향으로 구현하였습니다.