DB HW 6

2007-11791 이준행

1. Design Explanation

|  |
| --- |
| typedef struct{  long long index;  long long value;  } Dic\_element\_SSN;  typedef struct{  long long index;  char\* value;  } Dic\_element\_Other;  class Table{  public:  Table(void);  ~Table(void);  void \_UnitInsert(string f1, string f2, string f3, string f4, string f5, string f6);  void insert();  void makeDic(vector<char\*>& input, vector<Dic\_element\_Other>& target);  void DicToStore(vector<char\*>& temp, vector<Dic\_element\_Other>& dic, vector<long long>& store);  vector<long long>\* \_\_scan\_table(int fid);  vector<Dic\_element\_Other>\* Table::\_\_scan\_Other\_dic(int fid);  vector<Dic\_element\_SSN>\* Table::\_\_scan\_SSN\_dic(int fid);  void select(vector<int> fid, int column\_no, int oper, char\* value, ofstream& output\_file);  private:  vector<long long> SSN;  vector<Dic\_element\_SSN> SSN\_Dic;  vector<long long> SSN\_temp;  vector<long long> FirstName;  vector<Dic\_element\_Other> FirstName\_Dic;  vector<char\*> FirstName\_temp;  ……  char\* assignVarChar(string f, int \_size); |

Table Class는 SSN Column과 아닌 Column들에 따라 데이터 형태와 구현법이 조금씩 다르다. 주 Column Store는 index만 저장하기 때문에 long long으로 선언하였다.

Storage 변수들의 Name Convention은 다음과 같다.

|  |  |
| --- | --- |
| <Column\_name> | Main Column store |
| <Column\_name>\_Dic | Dictionary |
| <Column\_name>\_temp | Input 값에서 Dictionary를 만든 후 Column Store를 Index로 채울 때 원 입력 배열을 보존해주는 임시 Storage |

Dictionary를 만들기 위해선 index와 값을 모두 저장할 수 있는 구조체가 필요하다. 그 이유로Dic\_element\_SSN과 Dic\_element\_Other란 Structure를 만들었다. 저장하는 값이 숫자인지 문자인지에 따라 Dictionary Vector에 들어가는 구조체가 달라진다. 쉬운 구현을 위해서는 SSN Dictionary도 Character array로 만들어 쓰는 법이 있기는 하지만, 구현 난이도에 비하여 메모리 소비가 크므로, 번거롭더라도 두 속성을 따로 정의하기로 한다.

각 함수의 기능은 아래와 같다.

|  |  |
| --- | --- |
| \_UnitInsert | 파일에서 정보를 읽어와 원 배열 보존 벡터( <column\_name>\_temp )에 저장시킴 |
| makeDic | Column 속성이 Character array인 경우, 원 배열을 읽어 Dictionary를 만듦. SSN인 경우에는 함수화하지 않고 별도로 구현함 |
| DicToStore | Column 속성이 Character array인 경우, Dictionary를 읽어 원 배열을 참고하여 본 Column에 Index로 저장한다. 그리고 원 배열을 저장했던 temp vector를 지운다. |
| insert | 파일을 불러왔다면, makeDic과 DicToStore를 이용하여 Column Store 형태로 DB를 생성해 주는 함수 |
| \_\_scan\_table | Fid를 받아 원하는 column을 출력할 수 있도록 pointer를 반환하는 함수 |
| \_\_scan\_Other\_Dic | 원하는 column을 출력할 수 있도록 Dictionary의 Pointer를 반환하는 함수. Character array와 Long long array Vector 두 종류가 존재하기 때문에 둘을 분리하여 구현하였다. 이 부분은 Character array 함수이다. |
| \_\_scan\_SSN\_Dic | 위 함수와 내용은 동일하지만 return type때문에 따로 구현하였다. 이 함수는 Long long vector인 SSN을 위하여 구현된 함수이다. |
| select | \_\_scan\_table, \_\_scan\_Other\_Dic, \_\_scan\_SSN\_Dic을 이용하여 select문을 구현하게 해 주는 함수이다. |

그리하여 함수가 실행되는 흐름은 아래와 같다.

|  |
| --- |
| while(!insert\_file.eof()){  …  column\_store.\_UnitInsert(\_SSN, \_FirstName, \_LastName, \_DOB, \_Adress, \_city);  …  }  column\_store.insert(){  …  makeDic(FirstName\_sorted, FirstName\_Dic);  …  DicToStore(FirstName\_temp, FirstName\_Dic, FirstName);  …  }  column\_store.select(columnNumber, 0, GEQ, "2946584165915", output\_file){  …  reference = \_\_scan\_table(column\_no);  …  if (column\_no != 0) reference\_DIC = \_\_scan\_Other\_dic(column\_no);  else reference\_DIC\_SSN = \_\_scan\_SSN\_dic(column\_no);  …  } |

|  |
| --- |
| void Table::\_UnitInsert(string f1, string f2, string f3, string f4, string f5, string f6){  SSN\_temp.push\_back(\_atoi64(f1.c\_str()));  FirstName\_temp.push\_back(assignVarChar(f2, 30));  LastName\_temp.push\_back(assignVarChar(f3, 30));  DOB\_temp.push\_back(assignVarChar(f4, 8));  Adress\_temp.push\_back(assignVarChar(f5, 128));  City\_temp.push\_back(assignVarChar(f6, 2));  }  --------------------------------------------  In main.cpp usage  --------------------------------------------  while(!insert\_file.eof()){  getline(insert\_file, \_SSN, '|');  getline(insert\_file, \_FirstName, '|');  getline(insert\_file, \_LastName, '|');  getline(insert\_file, \_DOB, '|');  getline(insert\_file, \_Adress, '|');  getline(insert\_file, \_city, '\n');  column\_store.\_UnitInsert(\_SSN, \_FirstName, \_LastName, \_DOB, \_Adress, \_city);  } |

파일을 읽어오는 부분이다. 한 Row를 읽어 배열 순서를 보존하여 저장하는 <column\_name>\_temp에 저장하는 방법으로 구현하였다. 특이점은 없다.

|  |
| --- |
| void Table::insert(){  vector<long long> SSN\_sorted;  vector<char\*> FirstName\_sorted;    for (long long i=0; i< SSN\_temp.size(); i++) SSN\_sorted.push\_back(SSN\_temp.at(i));    sort(SSN\_sorted.begin(), SSN\_sorted.end());    long long index\_SSN = 0;  long long compare = SSN\_sorted[0];  Dic\_element\_SSN \_temp;  \_temp.index = index\_SSN;  \_temp.value = SSN\_sorted[0];  SSN\_Dic.push\_back(\_temp);  for (long long i=1; i < SSN\_sorted.size(); i++){  if (compare != SSN\_sorted[i]){  index\_SSN++;  Dic\_element\_SSN temp;  temp.index = index\_SSN;  temp.value = SSN\_sorted[i];  compare = SSN\_sorted[i];  SSN\_Dic.push\_back(temp);  }  }  SSN\_sorted.clear();    makeDic(FirstName\_sorted, FirstName\_Dic);    for (long long i=0; i< SSN\_temp.size(); i++){  for (long long j=0; j< SSN\_Dic.size(); j++){  if (SSN\_temp.at(i) == SSN\_Dic[j].value){  SSN.push\_back(SSN\_Dic[j].index);  }  }  }  for (long long i=0; i< SSN\_temp.size(); i++) SSN\_temp.pop\_back();  DicToStore(FirstName\_temp, FirstName\_Dic, FirstName);    return;  } |

Insert 부분이다. Dictionary를 편하게 구현하기 위하여 별도의 array를 만들고, Sorting된 데이터를 저장하는 데 사용한다. 함수 실행이 종료되면 함수 내에서 선언된 변수는 사라지고 메모리를 반납하는 성질을 이용하여 함수 내에서 sorted array를 잠시 저장할 vector를 만들었다. 그리고 Dictionary 제작 후 메모리 반납을 위하여 일일이 clear를 해 주었다. (하지만 반영이 되지 않은 듯 하다.)

Sort가 되어 있는 데이터라면 단순 Scan만으로 Dictionary 제작이 가능하다. 첫 줄은 미리 삽입하고, 그 다음 줄부터 매번 바로 전에 삽입한 데이터를 기억하여 현 데이터와 비교하는 간단한 알고리즘으로 Dictionary를 구현하였다. 메모리를 절약하는 방법이라면 Sort되지 않은, 원 배열 데이터를 Scan하여 Dictionary vector에 하나씩 집어넣고 Sort를 하는 방법이 있다. 하지만 그 방법으로 실행하면 데이터가 들어올 때마다 매번 Binary Search로 Distinct check를 한 후, Dictionary에 삽입했다면 Dictionary를 sort한 후, index를 다시 달아야 한다. 이렇게 되면 구현도 번거롭고, 시스템에 Overhead도 많이 걸리게 된다. 그리하여 메모리를 조금 희생하여 간단하고 빠르게 Dictionary를 제작하는 방법을 택하였다. Memory usage peak를 너무 크게 하지 않기 위하여 한 Column의 Dictionary 제작 후 바로 임시 배열 저장 벡터는 clear시켰다.

다음에는 index를 참고하여 Column store에 원 값 대신 index를 저장한다. 위 코드에는 매우 비효율적인 Linear Search로 구현되어 있지만, 실제 구현은 Binary Search로 하였다.

|  |
| --- |
| void Table::select(vector<int> fid, int column\_no, int oper, char\* value, ofstream& output\_file){  vector<long long>\* column1;  vector<long long>\* column2;  vector<long long>\* column3;  vector<Dic\_element\_SSN>\* dic1\_SSN = NULL;  vector<Dic\_element\_Other>\* dic1\_Other = NULL;  vector<Dic\_element\_SSN>\* dic2\_SSN = NULL;  vector<Dic\_element\_Other>\* dic2\_Other = NULL;  vector<Dic\_element\_SSN>\* dic3\_SSN = NULL;  vector<Dic\_element\_Other>\* dic3\_Other = NULL;  column1 = \_\_scan\_table(fid[0]);  if (fid[0] != 0) dic1\_Other = \_\_scan\_Other\_dic(fid[0]);  else dic1\_SSN = \_\_scan\_SSN\_dic(fid[0]);  switch (oper){  case 0:  …  case 1:  -----  for (long long i=0; i< column1->size(); i++){  if (reference\_DIC\_SSN != NULL && reference\_DIC == NULL){  if (reference\_DIC\_SSN->at(reference->at(i)).value == \_atoi64(value)){  if (dic1\_SSN != NULL && dic1\_Other == NULL){  output\_file << dic1\_SSN->at(column1->at(i)).value;  }  else if (dic1\_SSN == NULL && dic1\_Other != NULL){  output\_file << dic1\_Other->at(column1->at(i)).value;  }  output\_file << "\_";  if (dic2\_SSN != NULL && dic2\_Other == NULL){  output\_file << dic2\_SSN->at(column2->at(i)).value;  }  else if (dic2\_SSN == NULL && dic2\_Other != NULL){  output\_file << dic2\_Other->at(column2->at(i)).value;  }  output\_file << "\_";  if (dic3\_SSN != NULL && dic3\_Other == NULL){  output\_file << dic3\_SSN->at(column3->at(i)).value;  }  else if (dic3\_SSN == NULL && dic3\_Other != NULL){  output\_file << dic3\_Other->at(column3->at(i)).value;  }  output\_file << endl;  }  }  else if (reference\_DIC\_SSN == NULL && reference\_DIC != NULL){  if (reference\_DIC->at(reference->at(i)).value == value){  if (dic1\_SSN != NULL && dic1\_Other == NULL){  output\_file << dic1\_SSN->at(column1->at(i)).value;  }  else if (dic1\_SSN == NULL && dic1\_Other != NULL){  output\_file << dic1\_Other->at(column1->at(i)).value;  }  output\_file << "\_";  if (dic2\_SSN != NULL && dic2\_Other == NULL){  output\_file << dic2\_SSN->at(column2->at(i)).value;  }  else if (dic2\_SSN == NULL && dic2\_Other != NULL){  output\_file << dic2\_Other->at(column2->at(i)).value;  }  output\_file << "\_";  if (dic3\_SSN != NULL && dic3\_Other == NULL){  output\_file << dic3\_SSN->at(column3->at(i)).value;  }  else if (dic3\_SSN == NULL && dic3\_Other != NULL){  output\_file << dic3\_Other->at(column3->at(i)).value;  }  output\_file << endl;  }  }  }  break;  ---------------  case 2:  … |

Select 함수의 일부분이다. 어려운 구현은 아니지만, long long인 SSN과 char\*인 다른 인자와 구현 방식을 다르게 해야 하기 때문에 위와 같이 복잡한 형태를 가지게 되었다. 추후, template를 이용하여 쉽게 구현할 수 있는 방법이 있는지 찾아봐야겠다.

출력할 column 번호를 받아 해당 Column, Dictionary를 받아오는 vector pointer를 달고, enum으로 정의한 Operator를 이용하여 switch로 구현하였다.

먼저, Value의 Column이 SSN인지 그 이외 Character array인지 구별한 후, 다시 Column1, 2, 3 각각에 대하여

1. Performance Analysis
   1. Hardware infra
      1. CPU
         1. Intel i5-3210M 2.5GHz (Ivy Bridge, 64bit, For Mobile)
         2. 1 socket, 2 cores
         3. L1 64KB2, L2 512KB2, L3 3MB Shared
      2. RAM : 4GB
      3. OS : Windows 7 Ultimate K SP1
      4. Memory usage in idle status : 2.05GB
   2. Memory Consumed : 49MB
   3. Elapsed Time
      1. File Load : 28.725 sec
      2. Data -> Dictionary -> Mapping : 24.548 sec
      3. Select time : 10.218 sec
2. Discussion
   1. Binary Search vs Linear Search

앞에서 Binary Search로 구현하였을 때가 Linear Search보다 더 성능이 좋지 않게 나왔다고 하였다. 10K data를 기준으로, Linear Search로 Mapping하였을 때에는 3.2초가 소요되었는데, Binary Search로 구현하였을 때에는 0.3초가 소요되었다. O()와 O()과의 차이는 10K 데이터 기준으로도 10배 이상의 성능 차이가 났다.

* 1. Row Store vs Column Store

Row Store는 구현이 매우 간편한 반면 메모리를 많이 소비하는 단점이 있었다. 5M 데이터로 실행 시간은 60초가 걸렸던 반면, 메모리는 1099MB나 소비하였다. 반면, Column store는 3M 데이터 기준으로 실행 시간이 53초 반면, 메모리는 49MB만 사용하였다. 데이터 개수가 5M개와 3M개라는 차이는 있지만, 약간의 실행시간 희생으로 메모리를 획기적으로 줄일 수 있었다.

* 1. 효율적인 자원 사용을 위한 코드 개선법

전 시간에 사용하였던 B+Tree를 사용하면 Dictionary 제작 과정에서 원 데이터 배열을 보존하기 위하여 <Column\_name>\_temp 배열을 보존할 필요 없이 데이터가 들어오는 대로 배열을 저장하여 Dictionary를 저장하면 원 데이터 배열을 보존할 필요가 없이 Dictionary 구현이 가능하므로 Memory peak을 줄일 수 있다.