HW #7

2007-11791

이준행

1. Design

|  |
| --- |
| class Table{  public:  vector<column> column\_list;  Table(void);  ~Table(void);  void \_UnitInsert(string value, int col\_num, int size);  char\* assignVarChar(string f, int \_size);  void insert(int col\_num, vector<char> type);  void makeDic(vector<char\*>& input, vector<tempElement>& realDic, int col\_num, char type);  void DicToStore(vector<tempElement>& temp, vector<tempElement>& dic, vector<columnElement>& store, char type);  };  typedef struct{  long long index;  char\* value;  } tempElement;  typedef struct{  long long index;  long long valueIndex;  } columnElement;  class column{  public:  column(void);  ~column(void);  char type;  vector<columnElement> columndata;  vector<tempElement> Dic;  vector<tempElement> tempStoreOriginal;  }; |

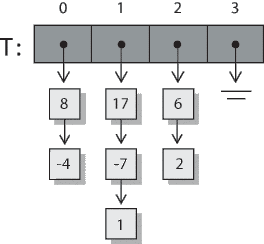
Column store design을 이번 과제에 맞게 변경하였다. 예전에는 Table에 column 이름을 할당하는 방식으로 storing을 했는데, 이번 Table은 Column들의 vector를 저장하는 형식으로 변경하였고, 대신 Column을 만들어 실질적인 data가 저장되도록 하였다 그리고 Column의 처음에 Type을 알 수 있도록 하여 추후 값을 Scan할 때, Column의 Data형을 미리 알 수 있게 구현하였다. 그리하여 각 Table마다 Column 수와 구조가 달라도 같은 Class로 처리를 할 수 있도록 구현하였다.

|  |
| --- |
| void Operator::selection(  Table\* output,  Table\* input,  vector<SelectCond> conds,  vector<int> projCols) {  char type = input->column\_list.at(conds[0].idxCol).type;  vector<long long> index\_list;  ClockChecker cck;  cck.SetStartClock();  switch (conds[0].op){  case EQ:  \_selection\_EQ(output, input, conds, projCols, index\_list, type);  break;  case NEQ:  \_selection\_NEQ(output, input, conds, projCols, index\_list, type);  break;  case LT:  \_selection\_LT(output, input, conds, projCols, index\_list, type);  break;  case LEQ:  \_selection\_LEQ(output, input, conds, projCols, index\_list, type);  break;  case GT:  \_selection\_GT(output, input, conds, projCols, index\_list, type);  break;  case GEQ:  \_selection\_GEQ(output, input, conds, projCols, index\_list, type);  break;  }  output->column\_list.resize(projCols.size());  for (size\_t j=0; j < index\_list.size(); ++j){  for (size\_t i=0; i < projCols.size(); ++i){  output->\_UnitInsert(input->column\_list.at(projCols.at(i)).Dic.at(input->column\_list.at(projCols.at(i)).columndata.at(index\_list.at(j)).valueIndex).value,i,25);  }  }  vector<char> type\_output;  for (int i=0; i< projCols.size(); ++i){  type\_output.push\_back(input->column\_list.at(projCols.at(i)).type);  }  output->insert(projCols.size(), type\_output);  cck.SetEndClock();  std::cout << "Select time : " << (double)cck.ElapsedClock()/(double)CLOCKS\_PER\_SEC << "sec" << '\n';  } |

Selection 구현 부분이다. Output Table에 먼저 공간을 할당하고 \_selection\_<operator> 함수들을 이용하여 조건에 해당하는 List를 만든다. \_selection\_<operator> 함수는 strcmp와 atoi 함수 등을 이용하여 조건값과 비교하는 형식으로 구현하였다. 만들어진 List 값에는 해당하는 record 일련번호가 저장되어 있다. 이 값을 참고하여 다시 Table을 구성한다. Table 구성에는 Column store에 구현하였던 \_unitInsert와 insert 함수를 사용하였다. 앞으로 구현할 함수도 조건에 해당하는 Column list를 뽑은 다음 bulk insert를 이용하여 Column store table을 구현할 것이다.

|  |
| --- |
| void Operator::join(  Table\* output,  Table\* lhs, Table\* rhs,  vector<JoinCond> conds,  vector<int> lProjsCols, vector<int> rProjCols) {  int size\_r = rhs->column\_list.at(conds[0].rhsIdx).columndata.size();  output->column\_list.resize(lProjsCols.size() + rProjCols.size());  vector<hash\_element> hash\_table;  hash\_table.clear();  for (long long i=0; i< size\_r; ++i){  hash\_element temp(-1, -1, '\0');  hash\_table.push\_back(temp);  }  // HASH TABLE FOR RIGHT TABLE  for (long long i=0; i< size\_r; ++i){  int coordinate = \_atoi64(rhs->column\_list.at(conds[0].rhsIdx).Dic.at(rhs->column\_list.at(conds[0].rhsIdx).columndata.at(i).valueIndex).value) % size\_r;  if (hash\_table[coordinate].index == -1){  …..  }  else{  …  }  }  //  // "size\_l" is NOT MODULATOR!!!!!!  // SCAN LEFT TABLE  for (long long i=0; i < lhs->column\_list.at(conds[0].lhsIdx).columndata.size(); ++i){  …  if (hash\_table[coordinate].value == l\_value){  …  }  else if(hash\_table[coordinate].index != -1 && hash\_table[coordinate].value != l\_value){  int ind = 0;  hash\_element\* chain = &hash\_table[coordinate];  do{  chain = chain->next;  if (chain->value == l\_value){  ind = 1;  break;  }  } while(chain->next != '\0');  if (ind == 1){  …  }  }  }  …..  } |

Join 부분이다. 과제에서 요구하는 Hash Join으로 구현하였다. Hash function으로는 table 크기를 이용한 mod 기법을 이용하였다. 메모리 공간을 희생하여 속도를 얻는 것이 Hash join의 특징이지만, Mod 기법은 메모리 할당량이 테이블 크기 이상으로 넘어가지 않는 것이 보장되는 기법이기 때문에 절충안으로 이 방법을 택하였다. 같은 mod값을 가지는 column index number는 linked list를 이용하여 처리하였다. 아래 그림에 대략적으로 알고리즘을 나타내었다.



한 Column에서는 값을 scan하면서 각 좌표에서 Table record number로 mod연산을 하고, 결과는 Hash table에 좌표와 좌표값을 적어넣는다. 다른 column에서는 scan하면서 똑같이 mod 연산 후, 같은 값이 있는지 살펴본다. 같은 값은 아니지만, 여러 값이 linked되어 있으면, link의 끝까지 검색하면서 자신과 같은 값이 있는지 살펴본다. 같은 값이 있으면 projection할 column을 수합하여 한 Record씩 column store에 집어넣는다. 모든 값이 다 모이면 insert 함수를 이용하여 Column store화 시킨다.

|  |
| --- |
| void Operator::orderBy(  Table\* output,  Table\* input,  vector<OrderCond> conds) {  // sort list by sort function  // use asc function (comparisonFuc) and desc function (will be a counterorder)  // sort(vector.begin(), vector.end(), conparisonFunc or countercomp);  // output as index order for keeping row  //  char type = input->column\_list.at(conds[0].idxCol).type;  ClockChecker cck;  cck.SetStartClock();  output->column\_list.resize(input->column\_list.size());  ….  switch(type){  case 'I':  sort(output->column\_list.at(conds[0].idxCol).tempStoreOriginal.begin(),  output->column\_list.at(conds[0].idxCol).tempStoreOriginal.end(),  comparisonFunc\_asc\_int);  break;  case 'F':  sort(output->column\_list.at(conds[0].idxCol).tempStoreOriginal.begin(),  output->column\_list.at(conds[0].idxCol).tempStoreOriginal.end(),  comparisonFunc\_asc\_float);  break;  case 'C':  sort(output->column\_list.at(conds[0].idxCol).tempStoreOriginal.begin(),  output->column\_list.at(conds[0].idxCol).tempStoreOriginal.end(),  comparisonFunc\_asc\_char);  }  }  else if (conds[0].desc == true){  switch(type){  case 'I':  sort(output->column\_list.at(conds[0].idxCol).tempStoreOriginal.begin(),  output->column\_list.at(conds[0].idxCol).tempStoreOriginal.end(),  comparisonFunc\_desc\_int);  break;  case 'F':  sort(output->column\_list.at(conds[0].idxCol).tempStoreOriginal.begin(),  output->column\_list.at(conds[0].idxCol).tempStoreOriginal.end(),  comparisonFunc\_desc\_float);  break;  case 'C':  sort(output->column\_list.at(conds[0].idxCol).tempStoreOriginal.begin(),  output->column\_list.at(conds[0].idxCol).tempStoreOriginal.end(),  comparisonFunc\_desc\_char);  break;  }  }  …  } |

Orderby는 C++ 라이브러리에 내장된 sort 함수를 이용하였다. 이 함수는 Merge sort로 구현되어 있어 O(n log n)의 수행 속도를 보장한다. 해당하는 Column의 값을 이용하여 Sorting한다. 이 때, Column에는 원 배열에서의 좌표값과 값이 같이 저장되어 있다. Sorting시에는 좌표값을 이용하여 Sorting이 되고, 그렇게 되면 좌표값의 정렬에 따라 index가 배열이 될 것이다. 그렇게 되면 다른 Column들은 그 값을 참고하여 재배열하게 된다.

정렬 이후에는 값을 이용하여 Projection할 Column을 찾아 한 줄씩 Column store화 시킨다.

|  |
| --- |
| output->column\_list.resize(groupCols.size() + conds.size());  …  for (int i=0; i< conds.size(); ++i){  switch(conds[i].op){  case COUNT:  aggr\_count(value\_aggr.at(i), value\_groupby.at(i), input, groupCols[0]);  break;  case MIN:  aggr\_min(value\_aggr.at(i), value\_groupby.at(i), input, conds[i].idxCol, groupCols[0]);  break;  case MAX:  aggr\_max(value\_aggr.at(i), value\_groupby.at(i), input, conds[i].idxCol, groupCols[0]);  break;  case AVG:  aggr\_avg(value\_aggr.at(i), value\_groupby.at(i), input, conds[i].idxCol, groupCols[0]);  break;  case SUM:  aggr\_sum(value\_aggr.at(i), value\_groupby.at(i), input, conds[i].idxCol, groupCols[0]);  break;  }  }  …  }  void aggr*\_<operator>(*vector<float>& vec, vector<char\*>& mem, Table\* input, int aggr\_col, int groupby\_col){  vector<int> count;  for (long long i=0; i < input->column\_list.at(groupby\_col).columndata.size(); ++i){  long long result;  if (mem.size() == 0){  mem.push\_back(input->column\_list.at(groupby\_col).Dic.at(input->column\_list.at(groupby\_col).columndata.at(i).valueIndex).value);  vec.push\_back((float)atof(input->column\_list.at(aggr\_col).Dic.at(input->column\_list.at(aggr\_col).columndata.at(i).valueIndex).value));  count.push\_back(1);  }  else{  result = binarySearch\_plan(input->column\_list.at(groupby\_col).Dic.at(input->column\_list.at(groupby\_col).columndata.at(i).valueIndex).value, mem, input->column\_list.at(groupby\_col).type);  if (result != -1){  vec.at(result) = (vec.at(result)\*count[result] + (float)atof(input->column\_list.at(aggr\_col).Dic.at(input->column\_list.at(aggr\_col).columndata.at(i).valueIndex).value)) / (count[result]+1);  count[result] += 1;  }  else{  vec.push\_back((float)atof(input->column\_list.at(aggr\_col).Dic.at(input->column\_list.at(aggr\_col).columndata.at(i).valueIndex).value));  count.push\_back(1);  }  }  }  } |

Group by, aggregation은 보통 Orderby를 이용하여 정렬한 다음, 같은 값끼리 모여있을 때, 간편한 Table scan으로 구현할 수 있을 것이다. 하지만, 이 방법은 O(n log n)의 Time complexity를 가지게 된다. 하지만 본인은 Table scan 한 번으로 Aggregation을 할 수 있도록 구현하였다.

Distinct한 value를 저장하는 저장소를 만든 다음, Table을 Scan하면서 distinct하면 새로운 원소로 저장하고, 중복이라면 값을 저장하는 vector 변수에 SUM, AVG, COUNT 등의 Aggregation을 수행한다. 이 방식으로 진행하면 별도의 Orderby없이 Group by를 수행할 수 있다.