CS 564 Project Stage 6

Minirel Query and Update Operators

Introduction

For this project stage, you will implement facilities for querying and updating Minirel databases. Specifically, you will implement selection, projection, insertion, and deletion. The parser that we will provide will parse the SQL-like commands and make the appropriate calls to the back-end.

SQL DML Commands

Minirel implements a simplified version of the SQL query language. The syntax of this language is described below using pseudo context-free grammar that is a continuation of the grammar described in Part 5 of the project. Recall that optional parts of statements are enclosed in square brackets.

A DML statement can either be a query or an update.

<DML_STATEMENT> ::= <QUERY>
| <UPDATE>

We now describe the format of a query.

<QUERY> ::= SELECT <TARGET_LIST> [into relname] FROM <TABLE_LIST> [WHERE <QUAL>]

The result of a query is either printed on the screen or stored in a relation named *relname*. To simplify the first case (i.e. print to the screen), the Minirel query interpreter creates a temporary relation which it then prints and destroys. Thus, in both cases, the name of a result relation is provided and will be passed to function SQ_Select (see below). If no qualification is specified, then the query simply prints the columns requested in the target list.

<TARGET_LIST> ::= (relname1.attr1, relname2.attr2, ..., relnameN.attrN)

This is a list of attributes attr1, attr2, …, attrN from relations relname1, relname2,... relnameN respectively that constitutes the target list of a query. All attributes in the target list should come from the same relation, except when the qualification is a join, in which case the attributes can come from the two relations begin joined . The "relnameN" qualifier can only be omitted if the query is on a single relation. It can be an alias of a relation if the alias is defined in TABLE_LIST. Please refer to <TABLE_LIST> (see below) for more of relation alias. The projection should be performed *on the fly* while the qualification is being evaluated, i.e. while the selection or join is being processed. You **should not** create a temporary relation with the result of the selection or join and then project on the desired columns. You do not have to worry about eliminating duplicates.

<TABLE_LIST> ::= (relname1 [alias1], relname2 [alias2],... relnameN [aliasN])

This list defines the target table list of a query. Aliases for each relation can be specified optionally. Once an alias is specified, it can be used as relation qualifier in TARGET_LIST and QUAL clause.

<QUAL> ::= <SELECTION>
|<JOIN>

Qualifications are very simple, either a selection or a join clause.

<SELECTION> ::= relname.attr <OP> value

A selection condition compares an attribute with a constant. The "relname" qualifier can be omitted if the query is on a single relation, or can be an alias if the alias is defined in TABLE_LIST clause.

<JOIN> ::= relname1.attr1 <OP> relname2.attr2

A join condition compares the join attribute value of one relation with another. The "relname" qualifier can be an alias if the alias is defined in TABLE_LIST clause. Non-equi joins can only be performed using the nested loops join method.

<UPDATE> ::= <DELETE>
| <INSERT>

An update is either an insert or a delete operation i.e. you cannot use Minirel to modify the value of an existing tuple in a relation.

<DELETE> ::= DELETE FROM relname [where <SELECTION>]

This operations specifies the deletion of tuples that satisfy the specified selection condition.

<INSERT> ::= INSERT INTO relname (attr1, attr2,... attrN) VALUES (val1, val2,... valN)

This will insert the given values as a tuple into the relation relname. Note that the values of the attributes may need to be reordered before the insertion is performed in order to conform to the offsets specified for each attribute in the AttrCat table. You can do this by using memcpy to move each attribute in turn to its proper offset in a temporary array before calling insertRecord.

Implementing the Relational Operators

For this part of the project you will need to implement the following routines. They will be called by the parser with the appropriate parameter values in response to various SQL statements submitted by the user. Thus you should not add or remove parameters unless you are prepared to modify the parser.

const Status QU_Select(const string & result, const int projCnt, const attrInfo projNames[], const attrInfo* attr, const Operator op, const char *attrValue)

A selection is implemented using a filtered HeapFileScan. The result of the selection is stored in the result relation called *result (a* heapfile with this name will be created by the parser before QU_Select() is called). The project list is defined by the parameters *projCnt* and *projNames*. Projection should be done on the fly as each result tuple is being appended to the

result table. A final note: the search value is always supplied as the character string *attrValue.* You should convert it to the proper type based on the type of attr. You can use the atoi() function to convert a char* to an integer and atof() to convert it to a float. If attr is NULL, an unconditional scan of the input table should be performed.

const Status QU_Join(const string & result, const int projCnt, const attrInfo projNames[], const attrInfo *attr1, const Operator op, const attrInfo *attr2)

Since time is short, you do not have to implement this operator. You might look at it to figure out how to implement select.C.

You only need to implement the standard tuple nested loops join algorithm. The two attributes being compared are *attr1* and *attr2*. As with the select operator, the result table will be created by the parser before QU_Join is called. The names of the two relations being joined can be found in the relName component of the two attrInfo[] parameters. If the two relations being joined have attributes with the same name, one of them will disambiguated using the suffix "_1" in the result schema. You should perform projection on the fly using information given in projCnt and projNames.

When implementing the nested-loops join algorithm use the following optimization. For each outer tuple, using the join attribute value of the outer tuple as the filter value start a filtered scan on the join attribute of the inner table. In this way the scan on the inner table will only return tuples that match the join attribute value of the outer tuple currently being processed. As each tuple from this scan is returned, produce an output tuple.

const Status QU_Delete(const string & relation, const string & attrName, const Operator op, const Datatype type, const char *attrValue)

This function will delete all tuples in relation satisfying the predicate specified by *attrName*, *op,* and the constant *attrValue*. *type* denotes the type of the attribute. You can locate all the qualifying tuples using a filtered HeapFileScan.

const Status QU_Insert(const string & relation, const in attrCnt, const attrInfo attrList[])

Insert a tuple with the given attribute values (in *attrList*) in relation. The value of the attribute is supplied in the *attrValue* member of the attrInfo structure. Since the order of the attributes in attrList[] may not be the same as in the relation, you might have to rearrange them before insertion. If no value is specified for an attribute, you should reject the insertion as Minirel does not implement NULLs.

Getting Started

Download the Zip file for this stage (your instructor will give the precise instruction). This Zip file includes an implementation of the solution to part 5. Here is a list of the key files you need to implement.

select.C - Stub for QU_Select. You have to implement this.

join.C - Stub for QU_Join. We already implemented this for you.

insert.C - Stub for QU_Insert. You have to implement this.

delete.C - Stub for QU_Delete. You have to implement this.

Testing

The test files are named qu.1, qu.2, etc. in directory "testqueries". Files qu.1, qu.5, and qu.7 test selection/project, insertion, and deletion without indexing. **At the minimum, your code should work for these files**. Feel free to augment these files and use the augmented versions to further evaluate your code.

Other files in "testqueries" test the above plus join, supposedly with indexing, except that we have turned off the indexing step for now. You can use them to further test your code.

You can run the above files by manually feeding the commands in the files to your code, or you can modify the script "qutest", and then use it to run the above files.
