The RQO Tool
Martin Estes
January 2016

 RQO is a tool that optimizes database queries and chooses the best plan (implementation) by comparing estimated costs of generated algorithms against each other. It then executes the cheapest-cost algorithm. This document explains the use of RQO in detail. RQO comes packaged with the DxTer environment. It was received from Bryan Marker, through GitHub.

 It is important to note that before coding anything in RQO, you must navigate to the layers.h file found in the src folder of DxTer and change the flag for RQO to 1 and the rest to 0.

# Relational Tables

## Schema Declarations

 Navigate to the folder within DxTer named RQO. It should be on the path ./src/RQO. Work takes place inside a file named userInput.cpp. In this file can be found the following functions: buildUserTables(), UserFunction(), BuildExampleTables(), and ExampleFunc(). The first two are labelled with comments and is the workspace for the user. The latter two are examples that may be referred to if help is needed, and can be run to see mock data work.

The first step to use RQO is to create tabular data. Consider a SQL table called Zipcodes:

Table 1 Relation Zipcodes

|  |  |
| --- | --- |
| **Zip** | **City** |
| 67226 | Wichita |
| 60606 | Fort Dodge |
| 50302 | Kansas City |
| 54444 | Columbia |
| 66002 | Liberal |
| 61111 | Fort Hays |

RQO operates in a C++ environment and follows standard C++ syntax. A table is a custom object of type **Relation**. The code that creates the table is:

Relation \*zipcodes = new Relation(“zipcodes”);

This should be coded in the function void buildUserTables() at the top userInput.cpp, which is provided with the RQO source distribution. In the code snippet above, a pointer to the object named zipcodes is created, as well as passing an argument of zipcodes[[1]](#footnote-1). This redundancy is a personal preference, as these two names do not need to match.

 So far, all that’s been created is an empty table. It still needs fields to define each column of data and then rows of tuples. To add a column of data, a relation has a function called:

addAttribute(string name, string type, Boolean indexable)

The *type* argument must be constrained to “number” or “string”. Argument *indexable* is true if this attribute of the relation is to be indexed. Adding columns to your relation looks like:[[2]](#footnote-2)

Example 1:
zipcodes->addAttribute(“zip”, “number”, true);
zipcodes->addAttribute(“city”, “string”, false);

## Tuple Declarations

 Next, tuples need to be created and stored. A tuple is instantiated separately and populated with its own set of data before being put into a table. The creation of a tuple is just:

 Tuple exampleTuple;

Tuple takes no arguments to be instantiated. It is created as an object, instead of a pointer to the object. This is because it is used here and nowhere else. To add fields of data to a tuple, use the Tuple method:

 addField(string name, string value)

The value should be a string, even if it is a number.

Tuple has a method to add this tuple to a table. This function is:

addTuple(Tuple tuple)

Example 2:
Tuple zip1;
zip1.addField(“zip”, “67226”);
zip1.addField(“city”, “Wichita”);
zipcodes->addTuple(zip1);

The above code would be repeated for every row in the relation.

 RQO supports empty fields within tuples. To represent this, using a line from above as an example, you would pass in an empty string as the argument:

zip1.addField(“city”, “”);

A tuple must have a value in place for each column in the table, even if that value is null. Use the empty string syntax above to insure that nulls are represented.

A Relation object, once it is completed, must be added into the list of Relation objects so that it can be used elsewhere. This list is called userRDB, which stands for user’s relational database. A Relation is added with the following call:[[3]](#footnote-3)

userRDB.push\_back(zipcodes);

Once all of your relation tables have been created, the main part of the coding can begin.

## A Summary of the DxTer Tool

Before writing queries, it is important to understand what DxTer does in order to understand how itworks.

DxTer transforms graphs. A directed graph, starting from what DxTer refers to as an input node, is formed using nodes to represent operations, and edges to represent streams of data. The graph then ends at an output node. An input node is special as it has no stream feeding it data. An output node is similar, though it is the opposite as it has no streams going to any other node. Every DxTer graph must have at least one input and one output node, and then any number of interconnecting nodes.

Each node in the graph, excluding the special inputs and outputs, has one or more stream inputs and has one or more stream outputs. If the number of inputs is incorrect, DxTer terminates execution and throws an error. The output of each node is copied to each output edge it has.

A DxTer graph must begin with an input node, but it may have multiple input nodes, as it may be necessary to retrieve data from multiple relations. This goes the same for output nodes. Each stream must end at an output node, or it is deleted during the optimization process.

DxTer uses this graphical organization to insure correctness. The optimizations it runs in the background are correct through refactoring, meaning that the refactorings of the graph are proven to maintain correctness, and therefore maintain the semantics of the original graph.

# Queries

## One Relation Queries

To represent a query, let’s assume the user database consists of the following relations:

Table 2: Relation Orders

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ono | cno | eno | received | shipped |
| 1020 | 1111 | 1000 | 10-DEC-94 | 12-DEC-94 |
| 1021 | 1111 | 1000 | 12-JAN-95 | 15-JAN-95 |
| 1022 | 2222 | 1001 | 13-FEB-95 | 20-FEB-95 |
| 1023 | 3333 | 1000 | 20-JUN-97 | null |

Table 3: Relation Odetails

|  |  |  |
| --- | --- | --- |
| ono | pno | qty |
| 1020 | 10506 | 1 |
| 1020 | 10507 | 1 |
| 1020 | 10508 | 2 |
| 1020 | 10509 | 3 |
| 1021 | 10601 | 4 |
| 1022 | 10601 | 1 |
| 1022 | 10701 | 1 |
| 1023 | 10800 | 1 |
| 1023 | 10900 | 1 |

Table 4: Relation Parts

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pno | pname | qoh | price | olevel |
| 10506 | Land Before Time I | 200 | 19.99 | 20 |
| 10507 | Land Before Time II | 156 | 19.99 | 20 |
| 10508 | Land Before Time III | 190 | 19.99 | 20 |
| 10509 | Land Before Time IV | 60 | 19.99 | 20 |
| 10601 | Sleeping Beauty | 300 | 24.99 | 20 |
| 10701 | When Harry Met Sally | 120 | 19.99 | 30 |
| 10800 | Dirty Harry | 140 | 14.99 | 30 |
| 10900 | Dr. Zhivago | 100 | 24.99 | 30 |

Consider the following SQL query:

**SELECT ono, cno, shipped
FROM orders
WHERE ono > 1000**

To translate this query into RQO format, create a retrieval operation for the specific relation. RQO has several forms of retrievals, but it decides which algorithm to use based on a cost estimate. The following code should all be written in the function UserFunction(), which is marked with a comment in the code.

 To create a retrieval operation, create a list of fields to be taken (a.k.a. “projected”) from the relation. These must be strings.

 **Example 3:**

 **set<string> AFields;**

 **AFields.insert(“ono”);**

 **AFields.insert(“cno”);**

 **AFields.insert(“shipped”);**

In addition to a list of fields, the Relation you are querying must be taken from the userRDB. A function is provided in order to do this called:

getRelationByName(string name)

This function returns a pointer to the Relation, so be sure to save it in a variable accordingly:

Example 4:
Relation \*orders = getRelationByName("orders");

Next, an InputNode is instantiated and saved as a C++ pointer. A retrieval in RQO is represented by one of DxTer’s input nodes. To instantiate one of these, use the following:

new InputNode(string name, string sortBy,
 set<string> fields, string tablename,
 string query)

where name is the name of the *InputNode*, *sortBy* is the field you wish it sorted on, fields is the set created in example 3, *tablename* is the name of your table, and the query is the where part of the SQL query.

Example 5:
InputNode \*inA = new InputNode(“orders”, “ono”, AFields,
 orders->getName(), “ono > 1000”);
inA->SetRelation(orders);

The lines after the instantiation must be included. The function:

SetRelation(Relation \*relation)

takes the pointer to the Relation table used and tells the InputNode where to get the information from. This is necessary since, as stated above, an input node cannot have input streams. Therefore, it needs to house the pointer to the relation directly in the object.

 To finish up, DxTer needs an output node. DxTer has an object called a poss, which is representative of a possibility.[[4]](#footnote-4) When a poss is created, it takes the final node in the user created structure, adds an output node to it, and then wraps the entirety of the graph in the possibility. In this case, the final operation is inA.

A poss is then wrapped in a RealPSet object, which gives it the functionality to be optimized. The syntax for these is standardized to the following:

Example 6:
Poss \*poss = new Poss(1, inA);
RealPSet \*pset = new RealPSet(poss);
return pset;

The return statement concludes the function UserFunction(), which is the function that all of the code for the SQL query has been written in.

Returning the RealPSet is the last part of the process that the user is responsible for. DxTer should run this code, optimize it, and produce all of the output with no further coding. Your query is complete.

 A complete list of RQO nodes and operations is included at the end of this document. This API includes all relevant functions for each node, as well as the node’s purpose. The next section uses one of these as an example. A tutorial on how to run the written code is included right before the API under the section *How To Run The Optimizer*. There is no point where the code that DxTer has generated will need to be run, as it is run automatically.



## Multi Relation Queries

Consider the following SQL query:

SELECT \*
FROM orders, odetails
WHERE ono > 1000 and orders.ono = odetails.ono

This query requires two separate retrievals that are combined via a join. Assume two InputNodes have been created following the procedure of the previous section. One is named inA and is the same as above, and the second is inB, which retrieves tuple data from odetails where ono > 1000.[[5]](#footnote-5)

 To join these tuple streams, a Join operation is added to the graph. A Join operation requires two vectors of type string to know which field to join on. The Join operation is then instantiated by the following function:

new Join(String sortBy, vector<string> in0Fields,
 vector<string> in1Fields)

where *sortBy* is the field we want the data sorted on, *in0Fields* are the fields on which the first input should be joined on, and *in1Fields* are the fields that the second input should be joined on. At this time, DxTer only allows joining on one field. The following example shows how to create a Join node with stream inputs inA and inB:

Example 7:
vector<string> joinFields0;
joinFields0.push\_back(“ono”);

vector<string> joinFields1;
joinFields1.push\_back(“ono”);

Join \*joinNode = new Join(“ono”, joinFields0, joinFields1);

A Join operation is the first node that has been created that is not a retrieval. Since the structure that is being built is a graph, DxTer needs to be told how to connect the operations. The function for doing so is:

AddInput(\*Node input, Integer outputNumber)

where *input* is an input stream, and *outputNumber* is the index of this output stream from the previous node. As stated above, a node can have any number of copies of its outputted data, which necessitates this number. Here’s how to add inputs to a Join:

Example 8:
joinNode->AddInput(inA, 0);
joinNode->AddInput(inB, 0);

 The query graph is concluded with another possibility structure:

Example 9:
Poss \*poss = new Poss(1, joinNode);
RealPSet \*pset = new RealPSet(poss);
return pset;

 This code mirrors the previous query’s poss code, except that join is now the final node in the graph structure and is passed into the poss. This is the final statement in the function UserFunction().

[[6]](#footnote-6)

Consider the following query:

SELECT \*
FROM orders, odetails, parts
WHERE ono > 1000 and orders.ono = odetails.ono and parts.pno = odetails.pno and parts.qoh > 4

This query is similar to the previous query shown, except that this time it has a third set of data that needs to be combined. It is not combined on the same field as the previous data has been either. All of that being said, there is nothing new that is needed in this query.

 For this example, assume that the previous query’s code has been written. That means InA, InB, and joinNode have been instantiated, and connected together. The code in Example 9 has not been written, however, as it is the final thing to be written for every query.

 The addition to this graph needs to be started with another InputNode, called InC for this example. This InputNode retrieves from the Relation ***parts*** all data contained in each field where the field qoh meets the requirement of qoh > 4. The fields are specified as they have been previously, with a set of input values given to the InputNode.

 The second join to be created, named joinNode2, has a syntax identical to the previous join made. The only difference here is the inputs and fields that it is joining over.

Example 10:
**vector<string> joinFields0;
joinFields0.push\_back(“pno”);

vector<string> joinFields1;
joinFields1.push\_back(“pno”);**

Join \*joinNode2 = new Join(“ono”, joinFields0, joinFields1); joinNode2->AddInput(joinNode, 0);
joinNode2->AddInput(inC, 0);

Note above that this join is the first one to have a different field to sort on than it is joining on. The only detail that may have a discrepancy is which field is joining into which. RQO’s joins the second input stream added to the first input stream. Switching the order of when each input is added changes which set of data is added into the other.

 Once this join is added, the standardized ***poss*** code can be added to the end.

Example 11:
Poss \*poss = new Poss(1, joinNode2);
RealPSet \*pset = new RealPSet(poss);
return pset;



Query Specification

 Here is the grammar of 1-relational queries that RQO can process:

**Predicate : AndClause | AndClause OR Predicate ;
AndClause : Clause | Clause AND AndClause  ;
Clause: Attribute Rel String ;
Rel : = | != | > | >= | < | <= ;
Attribute: FieldName ;**

 A query is composed of one or more *or* statements, each of which contains the conjunction clauses. As with the examples above, the predicate can consist of just one clause, without any *and* or *or* statements.

 A clause is an attribute and a string, connected by a relational operator. The relational operators can be equals, not equals, less than, greater than, less than or equal to, or greater than or equal to.

 The attribute in a clause is the name of a field in the relation that is being drawn from. It must match a field name, or an error is thrown. The string can be any string.

## How to Run the Optimizer

 At this point, your relations and queries should all be represented in code as demonstrated above. Once this is done, open up a terminal/command line. Navigate to the uppermost level of the DxTer folder in which the Makefile is located. You can navigate through the terminal using the command *cd [File Path to your DxTer program].*

Once you are in this folder, you can begin the process of running the code you have written, in order for DxTer to generate and run the optimized version it creates. To start with, you must compile this code. The Makefile has already been designed to compile all of the C++ code, including anything created by you. To invoke it, type in the command *make*.

The code is compiled and may take a while. If there is an error, it does not finish and reports where to find this error.

Common errors include:

* Typos
* Forgetting connections
* Using mismatched names between multiple nodes or relations
* Forgetting semi-colons
* Forgetting a parameter

Once the code has successfully compiled, there are a few options of commands you can perform. If you simply wish to see a list of arguments you may pass into the function, type in ./dxter.x. The arguments are as follows: 1, which runs the program you have written, and 2, which runs the example program provided.

Your code should run relatively quickly. It finishes by printing out the final output of all of your functions together.

If you wish to see all of the optimizations DxTer came up with, look in the CodeOutput.txt file in the folder you are currently in in the terminal. This text file is a list of refinements DxTer has made to your code to make it optimal, as well as their cost estimates.

**RQO Cost Estimates**

In order to optimize each graph, DxTer uses cost functions written in each node. For RQO, these cost functions are done in an estimate of Big Order notation for each function. After a Relation is given to an input node, every node further down the stream will refer to this specific Relation to make its estimate. For example, refer to the *Orders* Relation above. This specific relation has four tuples in it. Worst case, a scan will have to go through all of these tuples. If they are indexed, it would only have to go through the indexed tuples. Assume that the tuples are then sorted with a sort node further in the graph. This sort would assume the worst case of all four tuples would still be here, and would use n \* log(n), n being 4, to calculate its worst case.

 Once every node has calculated its cost, DxTer combines all of these into a total cost for each graph. It goes through every possible optimization, and compares the costs. It then picks the lowest cost as the final function that it will run.

 In order to optimize fully and check every possible function, there are some nodes that have a cost set at the maximum integer value for C++. An example of one of these nodes is the InputNode. It will be optimized to a retrieval operation during runtime. Due to DxTer’s functionality, there will be some versions of the graph in the CodeOutput.txt that still include these nodes and have an abnormally high cost. These may be ignored.

# RQO API

This section contains a list of all nodes and operations available for use in the RQO library, and how to call them and their methods.

## Nodes

**InputNode**. All RQO graphs must begin with this type of node. DxTer optimizes it to be one of multiple retrieval operations.

InputNode \*in = new InputNode(string name, string sortBy, set<string> fields, string filename, string query);

This method points the InputNode to a relation. This is required.

in->SetRelation(Relation \*relation);

**CrossProduct**. This node computes the cross product of two input streams. It requires two inputs.

CrossProduct \*cross = new CrossProduct(string sortBy);

**Join**. This node computes a join operation on two input streams. It requires two inputs. DxTer optimizes it into one of many join algortithms.

Join \*join = new Join(string sortBy, vector<string> in0Fields, vector<string> in1Fields);

**LeftOuterJoin**. This node computes a left outer join on two streams of data. It requires two inputs. This join operation is not a choice for optimization for a normal join, and must be created manually.

LeftOuterJoin \*ljoin = new LeftOuterJoin(string sortBy, vector<string> in0Fields, vector<string> in1Fields);

**RightOuterJoin**. This node computes a right outer join on two streams of data. It requires two inputs. This join operation is not a choice for optimization for a normal join, and must be created manually.

RightOuterJoin \*rjoin = new RightOuterJoin(string sortBy, vector<string> in0Fields, vector<string> in1Fields);

**OuterJoin**. This node computes a full outer join operation on two streams of data. It requires two inputs. This join operation is not a choice for optimization for a normal join, and must be created manually.

OuterJoin \*fjoin = new OuterJoin(string sortBy, vector<string> in0Fields, vector<string> in1Fields);

**Projection**. This node allows only tuples with specified fields through. It only has one input.

Projection \*proj = new Projection(string sortBy, set<string> &inFields);

**Sort**. This node sorts tuples based on a SINGLE field name. It only has one input.

Sort \*sort = new Sort(string sortBy);

**Trim**. This node removes entire columns of specified fields from the data. It only takes one input.

Trim \*trim = new Trim(string sortBy, set<string> &inFields);

**Union**. This node computes a union operation on two sets of data. It requires two inputs.

Union \*un = new Union(string sortBy, vector<string> inFields);

## Shared Methods

All nodes share these methods between them. This method connects a newly created node to its input.

Node->AddInput(Node \*input, Integer outputNumber);

This method returns the cost of a node, based on what it connects to in the graph. It is mostly used by DxTer for optimization purposes.

Node->GetCost();

## Relations

**Relation**. This object represents a table of tuples that are accessed as the data. It consists of a list of tuples.

Relation \*rel = new Relation(string name);

This method adds a column to the relation.

rel->addAttribute(string name, string type, bool indexable);

This method returns the name of the relation.

rel->getName();

This method returns the number of tuples in the relation.

rel->getSize();

This method prints out all of the data in the table.

rel->printTable();

**Tuple**. This object is a list of field value pairs. These pairs of data should be in the same order as that of the relation it is stored inside.

Tuple tup;

This method adds a field value pair to the tuple.

tup->addField(string name, string value);

This method removes a field from the tuple. Fields are expected to be unique.

tup->removeField(string name);

This method prints the contents of the tuple.

tup->printTuple();

1. In C++, a pointer is an object that “points”, or references, an object created in memory. [↑](#footnote-ref-1)
2. Pointers in C++ use a slightly different syntax than normal objects. The main difference being replacing the “.” with “->”. This arrow does essentially the same function. [↑](#footnote-ref-2)
3. The function “push\_back()” in C++ is a built-in method for vectors, which is a type of list. It places the object at the end of the list. [↑](#footnote-ref-3)
4. A **possibility** is a DxTer term that represents a set of concrete plans that are derived from an abstract plan. An abstract plan is a specification that is a dataflow graph of DxTer interfaces/operations. A concrete plan is a full or partial implementation of an abstract plan, containing one or more primitive (non-interface) operations. [↑](#footnote-ref-4)
5. The SELECT \* (or SELECT ALL) is accomplished by creating a vector of all fields in all FROM-listed relations. There is no shortcut in RQO for “SELECT ALL”. [↑](#footnote-ref-5)
6. The OutputNode depicted in the graph is created when the Poss object is created. [↑](#footnote-ref-6)