DG-Query: An XQuery-based Decision Guidance Query Language

Alexander Brodsky[1], Shane G. Halder[1] and Juan Luo[2]
[1]*Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, VA 22030, U.S.A.*
[2]*Information Technology Unit, George Mason University, 4400 University Drive, Fairfax, VA 22030, U.S.A.*

Keywords: DG-Query, XQuery, Mathematical Programming.

Abstract: Decision optimization is broadly used for making business decisions such as those for finding the best production planning in manufacturing. An optimization model may indicate the total cost of a certain supply chain given the various sourcing and transportation options used; the corresponding optimization problem can be to select among all possible sourcing and transportation options to minimize the total cost. Optimization modelling requires considerable mathematical expertise and effort to generate effective models. Additionally, the optimization process is heavily dependent on data. However, optimization languages such as IBM's ILOG CPLEX OPL and Bell Laboratories' AMPL, do not provide native support for manipulation of XML data. On the other hand, XQuery is a language for querying and manipulating XML data, which has become a ubiquitous standard (W3C) for data exchange between organizations; although, XQuery has no decision optimization functionality. To resolve this gap, this paper proposes DG-Query, an XQuery-based Analytics Language that seamlessly merges the XML data transformation and decision optimization capabilities. This is accomplished by first annotating existing XQuery expressions to precisely express the optimization semantics, and second to translate the annotated queries into an equivalent mathematical programming (MP) formulation that can be solved efficiently using existing optimization solvers. This paper presents DG-Query with an example, provides its formal semantics, and describes implementation through a reduction to MP formulation.

1 INTRODUCTION

In the age of data explosion, business intelligence analytics tools have been developed to provide users with the ability to gain insights about their business. Data analytics tasks can be classified into three main groups of (1) descriptive, (2) predictive and (3) prescriptive analytics. Descriptive analytics involves the manipulation and integration of large streams of data, using tools similar to database query languages. Predictive analytics acquires insights into the data using techniques of data mining and statistical learning. Prescriptive analytics deals with prescribing users actionable recommendations on how to move a (business) system towards an optimal outcome. This task is typically implemented using decision optimization tools.

Applications of decision optimization include deciding on business transactions within supply chains, stipulating environment policies which are aimed at public welfare, and finding the best response in an emergency. For example, given a repository of manufacturing plants and raw material suppliers, a business owner will need to decide on production planning and sourcing, namely, which products and in what quantities should be manufactured and where raw materials should be purchased from in order to maximize the total profitability.

Implementing solutions to such problems involves Mathematical Programming (MP) and/or Constraint Programming (CP), and using software packages called Optimization (MP or CP) Solvers. Optimization problems for both MP and CP are expressed by providing decision variables that range over some domain (e.g. reals, integers, binary, finite domain), constraints (e.g. arithmetic equations and inequalities over the decision variables), and the optimization objective (e.g. cost, revenue, profit, time) that needs to be minimized or maximized. Solving an optimization problem (either MP or CP) involves finding a value for each decision variable from its domain, in such a way that the optimization objective is minimized or maximized as required, while all constraints are satisfied (IBM, 2013).

Brodsky A., G. Halder S. and Luo J..
DG-Query: An XQuery-based Decision Guidance Query Language.
DOI: 10.5220/0004868201520163
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 152-163
ISBN: 978-989-758-027-7
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

While MP is geared more toward problems with numeric variables (over reals and/or integers), CP is mostly used for combinatorial optimization problems (where some variables range over finite domains), which are typical in applications for planning and scheduling (IBM, 2013). For both MP and CP, optimization modelling languages such as OPL (IBM, 2011), AMPL (AMPL, 2013), and GAMS (Boisvert, R.F., Howe, S.E., and Kahaner, D.K. Kahaner, 1985), are used to formulate decision variables, constraints and the objective function.

While MP and CP are most suitable technologies for finding high-quality solutions to optimization problems efficiently, the development of MP/CP models requires significant effort and mathematical expertise (in Operations Research) that most database application developers and data analysts do not have. Furthermore, the resulting models are typically not modular, extensible or reusable.

On the other hand, to support descriptive (as opposed to prescriptive) analytics tasks, database query languages such as XQuery can be easily used, which operates on XML (Extensible Markup Language) data. XML, designed by the World Wide Web Consortium (W3C), has become a standard for data exchange between organizations (W3C, Jan. 2012). Not only is data easy to represent in XML, XML is a self-describing language with the flexibility to define complex data structures. Furthermore, it has mechanisms to express constraints on the structures and contents of XML documents, such as XML schemas, DTDs and Schematron, which allows for rule-based validation regarding the detection of patterns in an XML document (Rick Jelliffe and Academia Sinica Computing Centre, 2002). The XML language is not complicated to use as a result of its simple constructs (W3C, Sep. 2006). Considering the widespread use of XML for data storage and exchange, it is no longer a nice feature to have but a necessity for any tool that is data driven. Not only should optimization modelling software support XML as a data source, the software should also provide an easy mechanism for querying XML data.

XQuery is an appropriate tool for the job. XQuery is also designed by W3C, and its language syntax for querying XML data is very similar to SQL for querying a relational database, making it easier to learn if the user already has knowledge of SQL. While XQuery is a fully-featured language, the FLWOR (For, Let, Where, Order by, and Return) expression provides an elegant way to query and manipulate XML data (W3C, Dec. 2010). Moreover, XQuery and XML are languages that are easy to learn and use by database application developers and data analysts. However, XML/XQuery does not support decision optimization.

Bridging the gap between the efficiency of optimization algorithms based on MP/CP and the ease of use by database application developers and data analysts using XML/XQuery is exactly the focus of this paper. We propose DG-Query, an XQuery-based Decision Guidance Query Language, which allows building optimization models by writing or reusing existing XQuery code/programs with minor annotations for optimization, thus making the language easy to use by database application developers or data analysts.

Seemless integration of the decision optimization models with XQuery programs presents a unique challenge. The reason for this is that optimization models declaratively express decision variables, constraints and the optimization objective, while XQuery programs are written as forwardly executed computation. We would like to avoid the direct encoding of optimization models, e.g. in XML, because this would create an *impedance mismatch*.

Instead, the idea of DG-Query is to annotate XQuery programs with *non-deterministic variables* to indicate that, intuitively, some values in the computation are unknown, and should be determined by the system, in such a way that a designated value (computed by the XQuery) be optimized subject to Boolean assertions (constraints), which are also added as program annotations.

The technical problem we need to overcome is to automatically translate DG-Query programs, with their *non-deterministic semantics,* into formal optimization models, expressed as MP or CP problems. The MP/CP problems are then solved to generate a solution to the optimization problem. The optimization solution provides values for non-deterministic variables, which makes the XQuery computation deterministic and allows an answer to be produced.

DG-Query is designed to extend the prevalent XQuery language with minimal annotation. As a result, we believe that DG-Query could be easily adopted by database application developers and data analysts especially if they are already familiar with XQuery. In summary, the contributions of this paper are:

- We introduce DG-Query, an XQuery-based analytics language for decision optimization and define its formal semantics
- We provide a reduction method to automatically transform DG-Query programs into formal MP

models, which can be solved by ILOG CPLEX solver
- We describe the implementation architecture of DG-Query

The rest of this paper is organized into 7 sections. In Section 2, we give a motivating running example. In Section 3, we briefly discuss the challenges of decision optimization, and discuss a few related works. Before we present the formal syntax and semantics of DG-Query, in Section 4, we use the running example to present DG-Query informally in an XQuery like syntax. We then give the formal syntax and semantics of DG-Query in Section 5. We present our implementation prototype and architecture in Section 6, discuss our experiment design in Section 7 and conclude with Section 8.

2 A RUNNING EXAMPLE

To make our discussion concrete, consider a simple example of decision guidance to support a small manufacturing network. Company X owns multiple manufacturing plants, building products to be sold. The raw materials necessary to build a product are provided by various suppliers. Each supplier has a limited quantity for the various raw materials. The company must stay below its overall raw material purchasing budget while maximizing its profit. The decisions that Company X needs to make are:
- The quantity of products each manufacturing plant should build
- The quantity of raw materials to be purchased from the various suppliers

Before explaining how DG-Query solves this problem, we describe a pure XQuery based solution in this section. The XML data representing the manufacturers, suppliers, and product parts are given below respectively. The XML schema definitions for manufacturers, suppliers, product parts are given in the Appendix.

Manufacturers XML Data:
```
<manufacturers>
  <products>
    <product mid='manuf1'
             prodid='product1'
             name='table'
             ppu='100.00' />
    <product mid='manuf2'
             prodid='product1'
             name='table'
             ppu='120.00' />
    ...
  /products>
</manufacturers>
```

The manufacturers XML data represents the manufacturing network. Each product has an associated manufacturer identifier ‘mid’, product identifier ‘prodid’, product description ‘name’, and the price to be sold per unit ‘ppu’.

Suppliers XML Data:
```
<suppliers>
  <parts>
    <part sid='supp1'
          partid='part1'
          name='wooden leg'
          cpu='2.50'
          supply='10' />
    <part sid='supp2'
          partid='part1'
          name='wooden leg'
          cpu='3.00'
          supply='5' />
    ...
  </parts>
</suppliers>
```

The suppliers XML data expresses the supply of raw materials available for purchase by manufacturers. Each part has an associated supplier identifier ‘sid’, part identifier ‘partid’, part description ‘name, the cost per unit ‘cpu’, and the available quantity ‘supply’.

Product Parts XML Data:
```
<productParts>
  <productPart prodid='product1'
               partid='part1'
               qty='4' />
  <productPart prodid='product2'
               partid='part1'
               qty='4' />
  ...
</productParts>
```

The product parts XML indicates the composition of a product by specifying the quantity of raw materials, i.e. parts needed to produce a piece of each product. The attribute ‘prodid’ corresponds to a manufacturer’s product, ‘partid’ corresponds to a supplier’s part, and ‘qty’ indicates the number of parts needed for the associated product.

With all the information given in the manufacturers, suppliers, and product parts XML data, we are trying to decide on the quantity of products for each manufacturer to produce and from which supplier to purchase the required parts, such that the objective, the total profit of manufacturers, will be maximized. The XQuery variable $manufItems_index (see below) is defined in the XQuery program to represent the quantity of each product to be produced by the manufacturer. In the variable $manufItems_index, the quantity is represented by the attribute ‘qty’, and the ‘prodid’ and ‘mid’ associate the product back to the original manufacturers XML. We have assigned a value to the quantities for each product. However, we need

to keep in mind that the set of quantities required to generate the maximum profit may not be known in advance and may be too complex to compute manually in real life.

```
 (: Manufacturer Product Quantity :)
let $manufItems_index  :=
  <products>
    <product prodid='product1'
             mid='manuf1'
             qty='0' />
    ...
    <product prodid='product3'
             mid='manuf1'
             qty='8' />
    ...
  </products>
```

Another XQuery variable $suppItems_index is defined to represent the quantities of raw materials, i.e. parts, purchased by manufacturers from each supplier. The quantity is represented by the attribute 'qty', and 'partid' and 'sid' associate the part back to the original suppliers XML. From the perspective of maximizing the total profit, the quantity of each purchased part from the suppliers is another decision to be made by the manufacturers. We assign a value for the quantities of parts but similarly, the set of quantities may not be known in advance in real life.

```
(: Supplier Part Quantity :)
let $suppItems_index  :=
 <parts>
 <part partid='part1' sid='supp1' qty='8' />
 <part partid='part2' sid='supp1' qty='0' />
 <part partid='part3' sid='supp1' qty='0' />
 <part partid='part4' sid='supp1' qty='3' />
    ...
 </parts>
```

To calculate the total profit for manufacturers, the following XQuery variables are defined. The variable $budget represents the total budget available for purchasing parts from suppliers. The variable $total_cost represents the total manufacturing cost of all manufacturers. The variable $total_revenue represents the total revenue of manufacturers by selling the products. The variable $total_profit represents the difference between the $total_revenue and $total_cost. The objective of this manufacturing model is to maximize the total profit, i.e. $total_profit.

```
(: Total budget available :)
let $budget as xs:decimal := 1000.00
let $total_cost as xs:double :=
  sum (
    for $s in $allSuppliers//supplier,
        $p in $allParts//part
    return
$suppItems_index//part[@sid=$s/@sid
       and @partid=$p/@partid]/@qty
           * $suppliers//part[@sid=$s/@sid
       and @partid=$p/@partid]/@cpu)
let $total_revenue as xs:double :=
  sum (for $m in
$allManufacturers//manufacturer,
        $p in $allProducts//product
return
$manufItems_index//product[@mid=$m/@mid
  and @prodid=$p/@prodid]/@qty *
$manufacturers//product[@mid=$m/@mid
   and @prodid=$p/@prodid]/@ppu )
let $total_profit as xs:double :=
            $total_revenue - $total_cost
```

To calculate the $total_profit, some intermediate variables are defined as follows for convenience.

```
(: Unique manufacturer ids :)
let $allManufacturers :=
  <manufacturers>
{
for $id in distinct-
values($manufacturers//product/@mid)
    return <manufacturer mid='{$id}' />
  }
  </manufacturers>

(: Unique supplier ids :)
let $allSuppliers :=
  <suppliers>
  {
    for $id in distinct-
values($suppliers//part/@sid)
    return <supplier sid='{$id}' />
  }
  </suppliers>

(: Unique product ids :)
let $allProducts :=
  <products>
{
  for $id in distinct-
values($manufacturers//product/@prodid)
    return <product prodid='{$id}' />
}
  </products>

(: Unique part ids :)
let $allParts :=
   <parts>
   { for $id in distinct-
values($suppliers//part/@partid)
     return <part partid='{$id}' />
   }
   </parts>

(: Expression: Qty Per Product Produced :)
let $producedqty :=
  <products>
  {
    for $p in $allProducts//product/@prodid
    return
      <product prodid='{$p}'
       qty='{
sum($manufItems_index//product[@prodid=$p]/@
qty)
       }'/>
  }
  </products>

(: Expression: Quantity Per Part Required :)
```

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
let $reqqty :=
<parts>
{
    for $p in $allParts//part/@partid
    return
      <part  partid='{$p}'    qty='{
sum($suppItems_index//part[@partid=$p]/@qty)
    }'/>
 }
</parts>
```

The XQuery variables $allManufacturers, $allSuppliers, $allProducts, and $allParts store a sequence of elements containing attributes corresponding to the unique identifiers of manufacturers, suppliers, products, and parts respectively. The variable $producedqty represents a sequence of elements containing attributes corresponding to a unique product and the total quantity of that product produced over the entire manufacturing network. And finally the variable $reqqty stores a sequence of elements containing attributes corresponding to a unique part and the total quantity of that part purchased from all the suppliers.

In this manufacturing network, we also want the system to enforce some business rules for the production process, e.g. the quantities of products produced by manufacturers must be greater than or equal to zero, and parts purchased by manufacturers from suppliers must also be greater than or equal to zero, and less than or equal to the quantity available by the suppliers. Additionally, the total cost of manufacturing must be less than or equal to the specified budget. The result of the business rules is represented by boolean variables with values of either 'True' or 'False'. The first two variables, $producedqty and $reqqty, are intermediate ones for business rule definitions. The translation of these business rules is straightforward and will be explained in Section 4.

```
(: Business Rule: Produced Quantity >= 0 :)
let $producedProductsGEZero as xs:boolean :=
  every $p in $manufItems_index//product
    satisfies xs:integer($p/@qty) ge 0

(: Business Rule: Purchased Parts >= 0 :)
let $purchasedPartsGEZero as xs:boolean :=
  every $p in $suppItems_index//part
    satisfies xs:integer($p/@qty) ge 0

(: Business Rule: Purchased Parts >=
Required Parts :)
let $purchasedPartsGERequiredParts as
xs:boolean :=
  every $p in $reqqty//part satisfies sum(
$suppItems_index//part[@partid=$p/@partid]/@
qty ) ge xs:integer($p/@qty)

(: Business Rule: Part Supply Per
Supplier >= Purchased Parts :)
let $partSupplyGEPurchasedParts as
xs:boolean :=
  every $s in $suppliers//part
    satisfies $s/@supply ge
   $suppItems_index//part[@partid=$s/@partid
        and @sid=$s/@sid]/@qty

(: Business Rule: Total Cost <= Budget :)
let $totalCostLEBudget as xs:boolean :=
                    $total_cost le $budget
```

The result of the XQuery program is the total cost, total revenue, and total profit generated based on the products produced and parts purchased, and the boolean values of the defined XQuery business rules and variables indicating whether the rules are satisfied. The result is returned in the format of a regular XML document.

Result XML:

```
 <result>
    <producedProductsGEZero>
    {
      $producedProductsGEZero
    }
    </producedProductsGEZero>
    <purchasedPartsGEZero>
    {
      $purchasedPartsGEZero
    }
    </purchasedPartsGEZero>
    . . .      . . .
    <totalCostLEBudget>
    {
      $totalCostLEBudget
    }
    </totalCostLEBudget>
    <total_cost>
    {
      $total_cost
    }
    </total_cost>
    <total_revenue>
    {
      $total_revenue
    }
    </total_revenue>
    <total_profit>
    {
      $total_profit
    }
    </total_profit>
</result>
```

3 RELATED WORK

DG-Query application development requires both XML/XQuery and Operations Research (OR) solutions. Difficulties arise when using OR tools (MP or CP) alone. The designing of the OR models requires full knowledge of the search space and the objective for the task of efficient system operations. The OR modelling abstraction (e.g., OPL (IBM,

2011) and GAMS (Boisvert, R.F., Howe, S.E., and Kahaner, D.K. Kahaner, 1985)) typically requires OR expertise, a skill that database/software developers may not have. Alternatively, XML and XQuery tools are more intuitive to IT professionals and have been widely used as a standard information exchange file format among heterogeneous systems, under multiple platforms. However, XQuery languages are not designed for decision optimization as they cannot express decision optimization problems, and further solve them. For continuous decision variables, uncountable possibilities of values to choose are obviously beyond the expressiveness of XQuery languages. For discrete cases, if the search space is large, it will be still inefficient for XQuery to try every possible discrete case. Its evaluation algorithms have not taken advantage of MP and CP search strategies to achieve potential efficiency and flexible optimization goals.

Optimization modelling software is heavily dependent on data, and a lack of native support for XML or XML processors requires that data stored or transmitted as XML be converted to a supported format. IBM's ILOG CPLEX Optimization Programming Language (OPL) software provides interfaces to various data sources including database support but not XML (IBM, 2011). Bell Laboratories' AMPL software falls into the same category as IBM's OPL, no XML support (AMPL, 2013). However, Maplesoft's Maple software does include an interface for XML. Maple incorporates an Extensible Stylesheet Language Transformations (XSLT) engine used to process XML data (Maplesoft, 2013). Although XSLT was created by W3C, its language constructs are verbose and complex, which could pose a challenge for use by OR analysts.

The language CoJava (Brodsky, A. and Nash H., 2006) offers both simulation of process modelling in Java, and the capabilities of true decision optimization. The syntax of CoJava is identical to Java but with a few special construct, i.e., definition of decision variables of a numeric value, the assertion of constraints and the designation of a variable as the objective to be optimized. The semantics of CoJava interprets the program as an optimal nondeterministic execution path. The constraints are part of the procedure. The procedure of the CoJava language can represent numerous modules of an entire model, or even larger, an entire software system. Running a CoJava program involves first finding an optimal execution path and then procedurally executing it.

Reporting applications over databases are intuitive and have long been established using the mature database query technology. A Decision Guidance Query Language framework (Brodsky A., Egge N. and Wang X.S., 2011)(Brodsky, A., Bhot, M.M., Chandrashekar, M., Egge, N.E., and Wang, X.S., 2009)(Brodsky, A., Egge, N.E., and Wang X.S., 2012) proposed a query language which was built on the Structured Query Language (SQL), and also included optimization functionality in it. It annotates existing SQL queries to precisely express the optimization semantics. It then translates the annotated SQL queries into equivalent MP, which can be solved efficiently. However, DGQL only supports the traditional relational database and does not support XML, a data format that gives more flexibility for data structures and is easier to be deployed in support of multiple platforms.

4 DG-QUERY VIA EXAMPLE

In this section, we use the existing XQuery statements in our manufacturing network example shown in Section 2 to illustrate the DG-Query language. We will discuss the DG-Query formal syntax and semantics in the next section.

Taking a pure XQuery program, only a few annotations are needed for software/database developers to re-write the XQuery program as a DG-Query program for decision optimization. In summary the annotations are (i) index set definitions; (ii) 'insert' statements with '?'; (iii) objective (maximize / minimize); and (iv) constraints.

Continuing with the example in Section 2, we assume that the XQuery variables, supplier part quantity $suppItems_index, and the manufacturer product quantity $manufItems_index, are decision variables to be determined, with the objective of maximizing the total profit described by the variable $total_profit. Instead of the given values for the XML attribute 'qty' for both XQuery variables $suppItems_index and $manufItems_index, we replace them with the following DG-Query statements.

```
(:Index set for all manufacturers'
products :)
let $manufItems_index as index(@mid,
@prodid) on $manufacturers :=
{ $manufacturers//product }

insert @qty as xs:integer on
$manufItems_index := ?

(: Index set for all suppliers' parts :)
let $suppItems_index as index(@sid, @partid)
```

```
  on $suppliers := { $suppliers//part }

insert @qty as xs:integer on
$suppItems_index := ?
```

The index(@attribute1, @attribute2, …, @attributeN) type is a special construct of DG-Query used to build an index set from a sequence of elements, where each element represents an object (i.e. a product) of the same type, as specified by an XPath expression over an XML document. The keys for the index set are indicated by the attributes @attribute1, @attribute2. …, @attributeN provided. The index set makes all of the attributes in the sequence of elements accessible to the optimization process.

The 'insert @qty as …' statement is another special construct of DG-Query that performs two operations: 1) For the optimization process, the attribute specified is marked as a decision variable to be solved; 2) After the optimization has completed, the solution for each decision variable is inserted as an attribute back into the source XML. Intuitively, with any assignment of nonnegative values to the attribute 'qty', there will be a total profit value given by the variable $total_profit. The task now is to automatically generate an assignment that gives the maximum profit for our manufacturing network example. We need to specify the objective explicitly as required by the optimization solver. The variable of $total_profit is calculated as the difference between the $total_revenue and $total_cost. The variable $total_revenue is calculated as the sum of cost per unit times quantities over the item set of $suppItems_index. The variable $total_cost is calculated as the sum of price per unit times quantities over the item set of $manufItems_index. We also defined a constraint, which is used to specify that $total_cost should be less than or equal to the overall budget indicated by $budget. Each constraint corresponds to a business rule specified in the pure XQuery program. The constraints assert a condition to be satisfied by the expressions.

```
constraint $producedProductsGEZero
constraint $purchasedPartsGEZero
constraint $purchasedPartsGERequiredParts
constraint $partSupplyGEPurchasedParts
maximize $total_profit
```

Given these annotations, DG-Query will parse the XQuery program and generate the necessary optimization model as well as convert the XML data into the appropriate format expected by the solver. After the solver has found a solution, as long as one is feasible, the values are written back into the source XML as shown below:

Updated Manufacturers XML data:

```
<manufacturers>
  <products>
    <product qty="0"
             mid="manuf1"
             prodid="product1"
             name="table"
             ppu="100.00"/>
    ...
    <product qty="18"
             mid="manuf1"
             prodid="product3"
             name="plate"
             ppu="10.00"/>
    ...
  </products>
</manufacturers>
```

Updated Suppliers XML data:

```
<suppliers>
  <parts>
    <part qty="10"
          sid="supp1"
          partid="part1"
          name="wooden leg"
          cpu="2.50"
          supply="10"/>
    <part qty="2"
          sid="supp1"
          partid="part2"
          name="wooden table top"
          cpu="70.00"
          supply="10"/>
    ...

  </parts>
</suppliers>
```

Note that the attribute 'qty' was not originally present in the source XML for either manufacturers or suppliers, and was added as a result of the optimization process.

5 FORMAL SYNTAX AND SEMANTICS

The DG-Query language is designed to extend the functionality of XQuery with annotations that can be used by an optimizer to solve decision optimization problems. The sections to follow describe the DG-Query annotations in detail.

5.1 Annotation Syntax

5.1.1 Index Sets

```
let $variable_name as index(@a1, ..., @a2)
  on $source_xml_variable_name := {
      XPath_Expression
   }
```

Index sets are used by the optimization process

to index integer or float arrays. They are also used to access XML element attributes that are required in the mathematical computation for the model. The XQuery variable representing an index set is linked to another XQuery variable containing the source XML over which to build the index. Both the index set variable and the source variable must be declared in the outermost scope of the program. Additionally the XML contained by the source variable must be well-formed. When using the index type annotation you must also specify the attributes that represent the unique key for identifying a particular element in the generated index. The XPath expression retrieves the sequence of elements of the same type from the source variable to be used for the index set.

5.1.2 Insert Expression

```
insert @attribute_name as [xs:integer |
xs:double]on $index_variable_name := ?
```

The insert expression is used by the optimization process to identify the variable that needs to be solved by the optimization solver. Note that the attribute name should not already exist in the source XML referred to by the index variable because it is inserted into the source XML after a solution has been found. The index set variable referenced must be declared beforehand. The attributes to be inserted/solved for can only be of integer or double types. The question mark assignment identifies that the value for the attribute is unknown and must be solved for during the optimization process.

5.1.3 Objective

```
[minimize|maximize] $math_expr_variable_name
```

The objective is a mathematical representation of the problem to be optimized. The keywords 'maximize' and 'minimize' are used to define the objective for the program. The variable name provided must contain a mathematical representation of the problem to be solved. The mathematical representation may be composed of previously declared intermediate expressions.

5.1.4 Constraints

```
constraint $boolean_rule_variable_name
```

Constraints are boolean expressions that define assertions about the data. The boolean expression must evaluate to true in order for the model to be solvable. The keyword 'constraint' is used to define a constraint annotation. The boolean variable referenced in the constraint must be previously declared. The optimization solver uses constraints to impose bounds on the solution.

5.2 Annotation Semantics

We now turn to define the semantics of DG-Query statements. As we described in previous sections, DG-Query is built upon XQuery, as it is a language extension of XQuery. We assume that XQuery semantics are kept and well understood. In addition to the semantics of XQuery, we formally define the aspect of DG-Query which is unique.

First we define the annotations of the DG-Query program. It contains any number of either index sets, 'insert' expressions with '?' assigned to it, constraints, and objectives. The constraints in DG-Query are equivalently a set of boolean variables defined in an XQuery program. Another assumption we claim is that the index set structure must be defined on an XQuery variable which has been defined on the outermost scope of XQuery programs. In this case, the index set will be assigned and evaluated correctly.

```
DG-annotation::= index set
  | insert statement with '?'
  | constraint
  | objective (minimize or maximize)
        ;
```

The input for a DG-Query program is a set of XML schemas S_i, $i = 1, 2, \ldots, n$, such as manuf.xsd, supplier.xsd, and productparts.xsd. The instances of XML schemas, d_i, $i = 1, 2, \ldots, n$, are regular XML documents, with the only exception that some values of attributes in XML documents are identified as missing, i.e., "?". The output of DG-Query optimization is an instance set d_i', $i = 1, 2, \ldots, n$, with all values populated for attributes.

$$DG_{program} := < S, XQuery, DG - annotation >$$
$$S := < S_1, S_2, \ldots, S_n >$$
$$DG - input := < d_1, d_2, \ldots, d_n >$$

A set of functions are defined as follows. The function f_o represents the objective function. This function maps from the independent variables, i.e., the values of attributes sets, which are annotated with missing values '?' to a real value output variable, i.e., the objective of DG-Query optimization. The function set f_j, $j = 1, 2, \ldots, k$, defines mapping relationships between input variables, i.e., the values of attributes sets, which are annotated with missing values '?', to a Boolean output variable. We assume there are k constraints defined in the overall DG-Query program. Each function f_j defines the mapping for the corresponding

constraint *j*.

$$f_0(d_1', d_2', \ldots, d_n')$$
$$f_1(d_1', d_2', \ldots, d_n')$$
$$\cdots$$
$$f_k(d_1', d_2', \ldots, d_n')$$

We use the set *All_I* to represent all possible instantiations of S_i, *i = 1, 2, …, n*. Another set *Feasible_I* represents the possible instantiations of S_i, *i = 1, 2, …, n*, which belongs to *All_I* and at the same time, satisfy all constraints *j, j = 1, 2, …, k*. Finally, two sets *Min_I* or *Max_I* are defined to represent the possible instantiations of S_i, *i = 1, 2, …, n*, which belongs to *Feasible_I* and at the same time, the value of objective function f_o over these specific instantiations is less than/greater than any other possible instantiations of *Feasible_I*.

$$All\_I(d_i, i = 1,2,\ldots,n) = \{(d_i', i = 1,2,\ldots,n) \mid d_i' \; satisfies \; S_i \wedge d_i' is \; an \; instantiation \; of \; d_i, i = 1,2,\ldots,n\}$$

$$Feasible\_I(d_i, i = 1,2,\ldots,n) = \{(d_i', i = 1,2,\ldots,n) \in All\_I | f_1(d_i', i = 1,2,\ldots,n) = True \wedge f_2(d_i', i = 1,2,\ldots,n) = True \wedge \ldots \wedge f_k(d_i', i = 1,2,\ldots,n) = True$$

$$Max\_I(d_i, i = 1,2,\ldots,n) = \{(d_i', i = 1,2,\ldots,n) \in Feasible\_I | \forall (d_i'', i = 1,2,\ldots,n) \in Feasible\_I (f_0(d_i'', i = 1,2,\ldots,n) \leq f_0(d_i', i = 1,2,\ldots,n))\}$$

$$Min\_I(d_i, i = 1,2,\ldots,n) = \{(d_i', i = 1,2,\ldots,n) \in Feasible\_I | \forall (d_i'', i = 1,2,\ldots,n) \in Feasible\_I (f_0(d_i'', i = 1,2,\ldots,n) \geq f_0(d_i', i = 1,2,\ldots,n))\}$$

Finally, we define the answer to the DG-Query program is an instantiation $d_i' \in Min\_I \; or \; Max\_I \; for \; i = 1,2,\ldots,n$.

6 IMPLEMENTATION PROTOTYPE AND ARCHITECTURE

6.1 Implementation Prototype

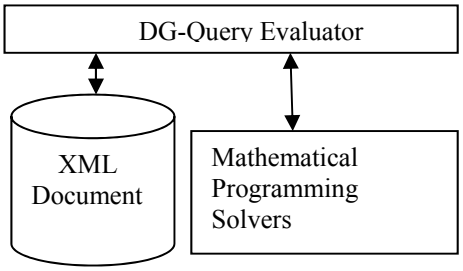
A prototype DG-Query compiler has been implemented in Java using the following tools:

- ANTLR (Another Tool for Language Recognition)
- BaseX XQuery 3.0 Processor
- IBM CPLEX OPL (Optimization Programming language) Java API

The ANTLR and BaseX libraries are written in pure Java allowing for seamless platform portability; however the IBM CPLEX OPL Java API requires native libraries so the corresponding platform specific software must be installed in order to run the compiler. The input into the compiler is an XQuery program annotated with DG-Query. The DG-Query compiler only contains a subset of the XQuery language and some syntax has been added to allow for more concise mathematical expressiveness.

6.2 Implementation Architecture

The key construction of DG-Query is an extension of the widely used XQuery language in two successive levels: Level 1 provides the ability to introduce non-deterministic variables (and their constraints) in an intuitive manner. The key construct is the type of DG-Query statement "`insert @qty as xs:integer on $manufItems_index := ?`", which essentially defines a decision variable, which is to be instantiated by solving a decision problem, and specifies constraints to be satisfied. After the definition, the user can refer to the variable in the conventional XQuery manner as if it has already been instantiated. Level 2 allows users to define an objective function over the decision variables using the construct "`maximize/minimize $objective`", with the objective value optimized. Mathematical programming concepts and technique are used behind the scene to instantiate the decision variables.



The conceptual system architecture is shown in the diagram above. In this demonstration, we use standard XQuery 3.0 (W3C), and ILOG/CPLEX as our mathematical programming (MP) solver. The DG-Query evaluator (*i*) compiles a DG-Query program, using data retrieved with XQuery from

XML documents, into an MP model expressed in OPL or a direct solver structure, (*ii*) uses an external MP solver to find a solution, and (*iii*) employs XQuery statement structure to insert the MP solution back into the source XML as a set of attributes.

7 EXPERIMENTS

We have taken the running example included in this paper and adjusted the syntax slightly to conform to the language grammar of the prototype DG-Query compiler. The program and input XML data was successfully translated into CPLEX OPL and solved accordingly with the following generated OPL Model:

```
tuple tuple_productParts_index {
      key string prodid;
      key string partid;
      int partqty;
};
      ... ...
Ordered                {tuple_productParts_index}
productParts_index  =  ...;
float budget = 204.00;
dvar int manufItems_index_qty[manufItems_index];
dvar int suppItems_index_qty[suppItems_index];
dexpr     int     product_index_producedqty[p     in
product_index] = sum(m in manuf_index)
       manufItems_index_qty[item(manufItems_index,
<m.mid,p.prodid>)];
       ... ...
dexpr   float   total_profit   =   total_revenue   -
total_cost;
maximize total_profit;
constraints {
       forall(mp        in        manufItems_index)
manufItems_index_qty[mp] >= 0;
       forall(sp        in        suppItems_index)
suppItems_index_qty[sp] >= 0;
       forall(part in part_index)
              part_index_reqqty[part]  <=  sum(s  in
supp_index)
   suppItems_index_qty[item(suppItems_index,
<s.sid,part.partid>)];
   forall(s in suppItems_index)
       suppItems_index_qty[s] <= s.supply;
total_cost <= budget;
}
```

The running time of mathematical programs is often sensitive to the way the problem is represented. For many classes of problems, a poor choice of mathematical representation will cause the solver to take significantly longer to find an optimal solution. One concern with DG-Query is that reduction process in modelling may add substantial overhead by introducing redundant or unnecessary variables and constraints. It is possible that a simpler or more concise modelling may lead to a significantly faster solution.

Our hypothesis is that problems described using DG-Query are solved as efficiently as when described using any other modelling tool. To test this, we took the manufacturing network problem from Section 2 and manually created a concise formulation of the OPL model. We then generated several instances of varying problem sizes and compared the execution time when solved using this OPL model and through DG-Query reduction. Although we did some preliminary experiments, the complexity of models can scale up. More advanced experiment designs will be conducted as part of future work.

8 CONCLUSION AND FUTURE WORK

This paper introduced the DG-Query language for decision optimization. We have defined its formal syntax and semantics. We also used an example to show that the overhead of using a high-level language is small when compared with manually crafted mathematical programming formulation. With the added benefit of having similar syntax to XQuery, we believe that DG-Query provides a practical solution for decision optimization. DG-Query can be used in many real industry applications such as finding the best course of action in an emergency, making a patient treatment decision for the best prognosis, deciding on public policies guided by the most positive outcomes, and deciding on business transactions within a supply chain, in the enterprise world.

Apart from providing native support for XML and XQuery, the DG-Query language should also promote modularity. The Process and Analytics Language (PAL) provides a great framework for building composite models from either atomic or other composite models (Shao G., Kibira D., Brodsky A., and Egge, N.E., 2011). The DG-Query language will use the PAL formalism for providing the necessary abstractions to support modularity.

ACKNOWLEDGEMENTS

This work is partially support by NIST funding.

REFERENCES

IBM, 2013. *Mathematical Programming vs. Constraint Programming*. Data Sources Retrieved from http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/mp-cp/

IBM, 2011. OPL, *the modeling language*: Data Sources Retrieved from http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r4/index.jsp?topic=%2Filog.odms.ide.help%2FOPL_Studio%2Fopllangref%2Ftopics%2Fuss_langref_datasources_1.html.

AMPL, 2013. http://www.ampl.com/

Boisvert, R. F., Howe, S. E., and Kahaner, D. K. Kahaner, 1985. *Gams: a framework for the management of scientific software*. ACM Trans. Math. Softw., 11(4):313–355.

W3C, Jan. 2012, *Extensible Markup Language (XML)*. Retrieved from http://www.w3.org/XML/

Rick Jelliffe and Academia Sinica Computing Centre, 2002, *The Schematron Assertion Language 1.6,* Retrieved from http://xml.ascc.net/resource/schematron/Schematron2000.html.

W3C, Sep. 2006. *Extensible Markup Language (XML) 1.1 (Second Edition)*. Retrieved from http://www.w3.org/TR/2006/REC-xml11-20060816/#dt-wellformed.

W3C, Dec. 2010. *XQuery 1.0: An XML Query Language (Second Edition)*. Retrieved from http://www.w3.org/TR/xquery/

Maplesoft, 2013. *Overview of the XMLTools Package*. Retrieved from http://www.maplesoft.com/support/help/Maple/view.aspx?path=XMLTools.

Brodsky, A. and Nash H., 2006. *CoJava: Optimization modeling by nondeterministic simulation.* In F. Benhamou, editor, Proceedings of Principles and Practice of Constraint Programming - CP, volume 4204 of Lecture Notes in Computer Science, pages 91–106. Springer.

Brodsky A., Egge N. and Wang X.S., 2011. *Reusing Relational Queries for Intuitive Decision Optimization*, System Sciences (HICSS), 44th Hawaii International Conference on, pages 1-9.

Brodsky, A., Bhot, M. M., Chandrashekar, M., Egge, N. E., and Wang, X. S., 2009. *A Decisions Query Language (DQL): High-Level Abstraction for Mathematical Programming over Databases*. Proceedings of the 35th SIGMOD International Conference on Management of Data.

Brodsky, A., Egge, N.E., and Wang X.S., 2012. *Supporting Agile Organizations with a Decision Guidance Query Language*, Journal of Management Information Systems 28 (4), 39-68.

Shao G., Kibira D., Brodsky A., and Egge, N. E., 2011. *Decision support for sustainable manufacturing using decision guidance query language*, International Journal of Sustainable Engineering 4 (3), 251-265.

APPENDIX

Manufacturers XML Schema Definition:

```
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="manufacturers">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="products"
                    minOccurs="1"
                    maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="products">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="product"
                    minOccurs="1"
                    maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="product">
    <xs:complexType>
      <xs:attribute name="mid"
                    type="xs:string"
                    use="required"/>
      <xs:attribute name="prodid"
                    type="xs:string"
                    use="required"/>
      <xs:attribute name="name"
                    type="xs:string"
                    use="optional"/>
      <xs:attribute name="ppu"
                    type="xs:decimal"
                    use="required"/>
      <xs:attribute name="qty"
                    type="xs:integer"
                    use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Suppliers XML Schema Definition:

```
<xs:schema

xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="suppliers">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="parts"
                    minOccurs="1"
                    maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="parts">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="part"
                    minOccurs="1"
                    maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="part">
    <xs:complexType>
      <xs:attribute name="sid"
                    type="xs:string"
                    use="required"/>
      <xs:attribute name="partid"
                    type="xs:string"
                    use="required"/>
      <xs:attribute name="name"
```

```
                    type="xs:string"
                    use="optional"/>
      <xs:attribute name="cpu"
                    type="xs:decimal"
                    use="required"/>
      <xs:attribute name="supply"
                    type="xs:integer"
                    use="optional"/>
      <xs:attribute name="qty"
                    type="xs:integer"
                    use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Product Parts XML Schema Definition:

```
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="productParts">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="productPart"
                    minOccurs="1"
                    maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="productPart">
    <xs:complexType>
      <xs:attribute name="prodid"
                    type="xs:string"
                    use="required"/>
      <xs:attribute name="partid"
                    type="xs:string"
                    use="required"/>
      <xs:attribute name="qty"
                    type="xs:integer"
                    use=" required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS