

Reusing Relational Queries for Intuitive Decision Optimization

Alexander Brodsky
Dept. of Computer Science
George Mason University
Fairfax, VA 22030, USA
brodsky@gmu.edu

Nathan Egge
Dept. of Computer Science
George Mason University
Fairfax, VA 22030, USA
negge@gmu.edu

X. Sean Wang
Dept. of Computer Science
University of Vermont
Burlington, VT 05401, USA
sean.wang@uvm.edu

Abstract—Decision optimization is used in many applications such as those for finding the best course of action in emergencies. However, optimization solutions require considerable mathematical expertise and effort to generate effective models. On the other hand, reporting applications over databases are more intuitive and have long been established using the mature database query technology. A decision optimization problem can be viewed as an “inverse” of the reporting problem. For example, a report may tell 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. Reusing existing reporting queries for decision optimization will achieve the dual goals of taking advantage of past investments and of making decision optimization more intuitive. To realize these goals, this paper addresses two related technical issues with a decision guidance query language (DGQL) framework. The first is to annotate existing queries to precisely express the optimization semantics, and the second is to translate the annotated queries into equivalent mathematical programming (MP) formulation that can be solved efficiently. This paper presents the decision queries with examples, provides formal syntax and semantics to DGQL, describes an implementation method through a reduction to MP formulation. Finally, the paper illustrates via experiments on a prototype system that the optimization tasks done with DGQL compete squarely with expertly generated MP models.

I. INTRODUCTION

An increasing number of decision-guidance applications are needed to provide human decision makers with actionable recommendations on how to move a complex process toward desirable outcomes. Examples include finding the best course of action in emergencies, deciding on business transactions within supply chains, and creating public policies aimed at most positive outcomes. The state-of-art implementation of decision guidance applications involves Mathematical and Constraint Programming (MP and CP), used for decision optimization, i.e., finding values for control variables that maximize or minimize an objective within given constraints. While software developers find database programming mostly intuitive, they typically do not have the mathematical expertise necessary for MP and CP. In addition, much investment has already been spent on database applications. Clearly, it is desirable to leverage this investment when building decision optimization applications.

In this paper, we describe a Decision Guidance Query Language (DGQL) framework. With DGQL, we provide a

step toward making decision optimization easier, especially in applications where database technology has been heavily used. In a nutshell, DGQL provides a query-like abstraction for expressing decision optimization problems so that database programmers would be able to use it without prior experience in MP, and more importantly, they would be able to reuse the queries already built into existing applications.

A key observation that motivates the DGQL development is that database languages are intuitive to use for reporting purposes, and decision optimization in principle is an “inverse” of much of the reporting functionality in place. In addition, code used in reporting functions often contains business logic that is needed in the decision optimization tasks. Based on this observation, a decision optimization problem in DGQL is written as a “regular” database program, i.e., a sequence of relational views and accompanying integrity constraints, together with some annotation of which database table column needs to be decided by the system (i.e., variables) and toward what goal (i.e., optimization objective). Here, existing queries in the reporting software can be directly used. Essentially, DGQL allows users to write optimization problem as if writing a reporting query in a forward manner.

The challenge in the DGQL approach, however, is how to execute the “inverse” decision optimization based on a “forwardly” expressed code. This is done by encoding the DGQL queries as an MP formulation, and by solving the MP problem and then deriving the solution to the DGQL optimization problem. A technical question with two interrelated parts arises: (1) is it possible to encode DGQL query as an MP formulation such that (2) the solution can be found efficiently? In this paper, we answer this question positively. Based on this answer, we believe that DGQL has the potential to achieve the best of both easy development and efficient systems for decision guidance. In terms of efficiency, the overall performance of intuitive DGQL queries compare squarely with expert-generated MP problems, as we shall show in the experimental section.

A common problem faced with the proposal of any new framework is acceptance and usage by the greater research community. By design, DGQL extends the most broadly used database query language SQL and adds only a minimal annotation which allows translation of SQL queries into optimization problems. We believe that this approach has a good chance of being broadly used exactly because it does

not require database programmers to learn a new language.

In summary, the contributions of the paper are:

- We introduce the DGQL database language for decision optimization and define its formal syntax and semantics.
- We provide a reduction method to automatically transform DGQL programs into formal MP models.
- We report experiments to show the efficiency of a DGQL implementation against expert-generated models.

The rest of this paper is organized into 7 sections. In Section II, we give a motivating, running example. In Section III, we briefly survey the state-of-art of decision optimization, and point out a few related works. Before we present the formal syntax and semantics of DGQL, in Section IV, we use the running example to present DGQL informally in a SQL-like syntax. We then give the formal syntax and semantics of DGQL in Section V. An implementation of DGQL via translation to MP is given in Section VI. Finally, we present our experimental results in Section VII, and conclude with Section VIII.

II. A RUNNING EXAMPLE

To make our discussion concrete, consider a simple example of decision guidance to support an emergency response (ER) officer who commands a relief operation for a natural or man-made disaster. The ER officer is in charge of providing ER locations with items such as foods, water, blankets, medical supplies, etc. Let us assume that a database is set up with the following tables.

```
demand(erloc, item, requested):
    a tuple indicates that for an ER location (erloc),
    the quantity requested of item is needed.
supply(vendor, item, price_per_unit, available):
    a tuple indicates that a vendor can provide
    at most available quantity of item at the
    price_per_unit.
orders(vendor, item, erloc, quantity):
    a tuple indicates the order of item in quantity
    quantity from vendor for location erloc.
```

With the above tables, a reporting system may have the following query to provide (total) cost information:

```
CREATE VIEW total_cost AS
SELECT SUM(s.price_per_unit*o.quantity) AS total
FROM orders o INNER JOIN supplier s
ON o.vendor=s.vendor AND o.item=s.item;
```

The above assumes that the item orders are decided elsewhere, and the database is simply to record these orders. Now consider the situation that the ER officer needs to decide these orders, with the objective of minimizing the cost. This is a typical “inverse” problem of the above reporting function. More specifically, we need to assume that the `orders` table needs to be filled such that all the demands are satisfied while the total cost is minimized. This is a non-trivial problem especially when there are constraints like limiting the number of vendors to supply a particular location (for the purpose, e.g., limiting the possible vendor contact points to ease the management and tracking).

State-of-art solutions will need to model the situation in a separate decision optimization system to make the order decisions. Our approach is to reuse the above SQL query, plus some auxiliary SQL statements and SQL-like annotations (e.g., for adding the constraints), to automatically generate mathematical programming models that will generate the optimal orders. The resulting annotated SQL query is called a DGQL query. We deliberately made the above query rather simple for easy presentation. One may imagine that more tables may be involved and some investment has already been made to debug and test the queries to make the business logic correctly reflected. This ability to leverage existing investment in database design through annotation is a distinct advantage over modelling languages like AMPL, GAMS, OPL and others, because it allows a clear separation between modelling the business objects and processes, and indicating *which* parameters are to be used for decision making. One can imagine a multistage decision optimization system where the output of one decision problem is used as the input to another. By allowing for heavy query reuse, new optimization problems can be created by a few annotations with little advanced planning.

Furthermore, additional information may be added to the system at different times, such as shipping options, packaging options, etc. Consider shipping options. As multiple shipping companies may be available with different pricing structure, it is useful to optimize the total shipping cost with the best shipping option. We may have three additional tables: `shipping_options` and `shipping_orders`. An additional query can be added to provide total shipping cost. (Care needs to be taken to make the item `orders` and `shipping_orders` consistent, perhaps by using integrity constraints in the database.) An interesting point here is that relational database systems allow modularized development, which increases the productivity of the programmers and makes the system development more intuitive.

With the additional needed information, the ER officer needs also make shipping (and other) decisions so that the total cost of items and shipping is minimized. Other requirements may come in as well, such as the time of delivery requirements. Again we may “invert” the reporting function to make decision optimizations, by reusing the SQL code already developed. If shipping option is added to the item `orders`, two tables, namely item `orders` and shipping `orders`, need to be filled to make the total cost optimal. The DGQL query will be modular in the same way that SQL is.

III. RELATED WORK

To build decision guidance applications, like the one in the previous section, requires both DBMS and operations research (OR) solutions. Using OR tools such as MP and CP poses significant challenges. Among them is the requirement of full knowledge of the search space and the objective for the task on hand for efficient system operation. The OR modeling abstraction (e.g., AMPL [1] and GAMS [2]) typically requires OR expertise, which makes the development by IT professionals challenging. In contrast, DBMS tools are

more intuitive and have been adopted by many application domains. However, DBMS query languages are not designed for decision optimization as they cannot express decision optimization problems, notably over continuous variables. Indeed, in the continuous variable case, there are potentially infinite possibilities to choose from that cannot be expressed as regular database query problem. For the discrete case, when potential choices are from a large space, populating tables with all possible choices and then performing a rank query can be quite inefficient. Although query languages can handle some limited discrete optimization computations, e.g., find a tuple that has a minimal value over a finite set of discrete choices [7], even in the cases of expressible rank queries, evaluation algorithms have not typically taken advantage of MP and CP search strategies to achieve potential efficiency and flexible optimization goals. More important, the optimization query will look very different from the reporting query, increasing the complexity of system management.

Specialized optimization tools (e.g., for optimizing price-revenue, transportation, sourcing, production planning, etc.) have been developed to be ready for end users, or integration with other systems (e.g., ERP/ERM), so that operations research expertise would not be necessary. However, this approach is not extensible, and does not support general decision-guidance application development.

The languages most closely related to DGQL are those that translate procedural algorithms into declarative constraints. These languages unify procedural and constraint semantics, such that the same program statements determine both interpretations. The language Modelica [5] supports unified models, which can define both simulation and optimization problems. Modelica models are translated into equations, which may in turn be solved by an optimizer or sorted and compiled into an efficient sequential procedure. Within pure functions (functions without side effects), Modelica can also translate procedural algorithms into constraint equations. Within these functions, Modelica gains the advantage of specifying constraints using familiar procedural operations and flow of control. However, Modelica is fundamentally an equational language, and it supports procedural algorithms only in this limited context.

The language CoJava [3] has a thoroughly procedural object oriented syntax and semantics of Java, and adds the optimization semantics of an optimal non-deterministic execution path. Thus, CoJava presents the developer with no visible boundary between procedures and constraints. Familiar procedural operations and flow of control can be used uniformly throughout an entire model, or even throughout an entire software system.

DGQL is in the spirit of Modelica and especially CoJava to minimize the learning curve for database application developers and to take advantage of existing database queries by conforming to a well understood syntax and semantics. While we chose SQL as the base language to be annotated, the same idea of annotating a database program and translating it to a formal optimization problem can be

applied to other database languages including Query-By-Example (QBE) and object relational query languages such as Hibernate Query Language (HQL). Using this approach, DGQL gives database application developers the flexibility to move model components freely back and forth between query evaluation and declarative optimization models.

Systems are being built for decision guidance in face of the challenges mentioned earlier, but the development of decision-guidance systems to-date amounts to a large modeling and software development effort. To complicate matters, additional development will periodically be needed when different types of new decision optimization requirements arise. DGQL, on the other hand, continues the tested database tradition in building a level of abstraction that is closer to the user intuition. By doing so, many applications can take advantage of decision optimization without large upfront investment and heavy long-term maintenance expenses.

IV. DGQL VIA EXAMPLE

In this section, we use the existing queries in our emergency response example shown in Section II to illustrate the DGQL framework. We will discuss the DGQL formal syntax and semantics in the next section.

Continuing with the example in Section II, we assume that the `orders` table is the one we need to fill, with the objective of minimizing the total cost as described by the `total_cost` query (view). Instead of the given database `orders` table, we replace it with a view created by the following SQL statement plus an augment annotation:

```
CREATE VIEW orders AS AUGMENT
  (SELECT d.erloc, s.vendor, s.item
   FROM demand d
   LEFT OUTER JOIN supplier s
   ON d.item=s.item)
WITH quantity INTEGER >=0;
```

Note here the nested query is a standard query, giving all possible pairings of vendor with item for each emergency location, while the view creation here is a DGQL annotation that indicates the resulting table of the nested query to be *augmented* with an extra column called `quantity` with the condition that the quantity has to be a non-negative integer. The augmented column is the one to be filled by DGQL execution automatically with optimal objective (to be annotated below).

Intuitively, with any assignment of nonnegative values to the augmented `quantity` column to the generated `orders` table, there is a total cost value given by the `total_cost` view. The task now is to automatically generate an assignment that gives the minimum total cost for our emergency example. However, if we examine the `total_cost` view, we note that table `demand` is not used. When the reporting query is written, we have assumed that the `orders` table is given externally and that all the demands are satisfied by the `orders`. This latter assumption can be checked by the database with the following constraint.

```
CREATE VIEW requested_vs_delivered AS
  SELECT o.erloc, o.item,
         SUM(o.quantity) AS delivered, d.requested
  FROM orders o
```

```

INNER JOIN demand d
ON o.erloc=d.erloc AND o.item=d.item
GROUP BY o.erloc, o.item, d.requested
CHECK delivered>=requested;
    
```

This constraint checks if an item request is satisfied at an emergency location with enough orders by comparing the total quantities of an item ordered for the location with the demand quantity at that location. This constraint needs to be explicitly given in order to generate `orders` table correctly, and DGQL supports the above constraint specification (while implementation of such a constraint in standard DBMS is not uniformly present).

Finally, we need to specify the objective explicitly. In DGQL, we use the following statement:

```

MINIMIZE total_cost;
    
```

Note that `total_cost` is the view defined in Section II. The semantics of the `MINIMIZE` is to create a table for `orders` (in general, all augmented tables are instantiated), in which the last column is filled with integer values, in the way such that the total cost is minimum among all possible orders to satisfy all the emergency demands.

The above optimization problem of minimizing the cost can be solved rather easily since the optimal strategy is to always to use the least-costly supplier for each item. This strategy can be coded directly in SQL and provide optimal solution without using DGQL or MP (although the code will look differently from the reporting query, increasing the management complexity). However, in general the capability of SQL in providing decision optimization is limited. For example, consider the situation that (1) each vendor has a limited supply of each item, and (2) in practice, we may want to limit the number of (say up to 3) vendors to supply all the items for a particular emergency location in order to facilitate easier tracking and management. In this case, it is not clear how to use SQL to directly code an optimal solution. In DGQL framework, this can be done with the following two additional constraints:

```

CREATE VIEW purchased_vs_available AS
SELECT o.vendor, o.item,
SUM(o.quantity) AS purchased, s.available
FROM orders o
INNER JOIN supplier s
ON o.vendor=s.vendor AND o.item=s.item
GROUP BY o.vendor, o.item, s.available
CHECK purchased<=available;
    
```

```

CREATE VIEW vendor_restriction AS
SELECT erloc, COUNT(*) AS vendors
FROM (SELECT DISTINCT erloc, vendor
FROM orders o
WHERE quantity>0)
GROUP BY erloc
CHECK vendors<=3;
    
```

The first constraint above states that the ordered item quantity from a vendor does not exceed its available quantity, and second enforces that the total number of distinct vendors for each emergency location does not exceed 3. With the above constraints, DGQL will find the optimal solution satisfying these constraint.

Note that a constraint is quite different from a selection (WHERE) condition. Indeed, in our example, constraints

will limit the possible `orders` table instantiations to be considered in obtaining the optimal solution by rejecting those that do not satisfy the constraints.

In addition, DGQL allows modular development. If shipping and other options are to be considered, we only need to annotate the relevant queries and add appropriate constraints in order to extend the decision optimization problem. DGQL also takes the relevant existing SQL queries without any change in participating in decision optimization, reusing existing investment.

V. FORMAL SYNTAX AND SEMANTICS

We now give a precise exposition of the DGQL syntax and semantics. The philosophy of DGQL is the add annotation, in the form of operators, to the standard SQL in order to express decision optimization problems. Like the standard SQL, the precise syntax and semantics is best explained in the relational algebra. Hence, in this section, we use the approach of adding a number of operators to the standard relational algebra. The informal, SQL-like expressions directly correspond to these operators.

Like the standard relational algebra, we assume a given set of base relations in a database, each having a unique name and schema. We also assume there is a supply of globally unique relation names to be used for assignments, each with a schema. We assume the conventional notions and notation (e.g., [6]), including those of schemas, domains, tuples, relations, databases and all the traditional algebraic operations, as well as the assignment statement (or view definition). We also use a simple form of aggregation (as the “aggregation formation” in [9]).

The annotation of DGQL consists of three operations: augmentation, constraint, and optimization. All three operators are used in the example in the previous section. Honoring the tradition, except for the optimization operator, which is the last statement in any DGQL program, we define augmentation and constraint as relational operators that can appear at any step of a relational expression.

For the sake of completeness, we list all the operators that can be used in DGQL. More precisely, we define DGQL query expressions recursively as follows:

- 1) (Base case) Each relation name (base relation or not) is a DGQL query expression; The schema of the expression is exactly the relation schema of the base relation;
- 2) (Traditional operators) Given DGQL query expressions e_1 and e_2 , then $\pi_L(e_1)$, $\sigma_C(e_1)$, $e_1 \times e_2$, $e_1 \cup e_2$, $e_1 \cap e_2$, and $e_1 - e_2$ are all DGQL query expressions; The operations are all the traditional relational operators, and we assume the schema requirement of the operators are satisfied by e_1 and e_2 ; The schemas of the expressions are as defined traditionally;
- 3) (Arithmetic) Given DGQL query expression e , then $\gamma[B = arith(L)](e)$, where L is a subset of numeric attributes in e , B is a new attribute, and $arith(L)$ is an arithmetic expression using attributes in L .

- 4) (Aggregation) Given DGQL query expression e and assuming A_1, \dots, A_n and B are attributes of the schema of e , then $aggr[A_1, \dots, A_n, f(B)](e_1)$ is a DGQL query expression, where f is an aggregation function (mapping bags to values); The schema of the expression is (A_1, \dots, A_n, fB) ; A special case is when $n = 0$, in which only fB attribute exists for the schema of the expression.
- 5) (Augmentation from Relation) Given DGQL query expressions e_1 and e_2 and assuming A_1, \dots, A_n are attributes in e_1 and not in e_2 , $\alpha[A_1, \dots, A_n \in e_1](e_2)$ is also a DGQL query expressions; The schema of the expression is the schema of e_2 extended with A_1, \dots, A_n ;
- 6) (Augmentation from Domain) Given DGQL query expression e , $\alpha[A \in domain](e)$ is also a DGQL expression, where $domain$ is a set of values of a specific type, e.g., $real[0, \infty)$ for non-negative reals. This form of augmentation is equivalent to the above one if $domain$ is taken as a (possibly infinite) relation with one attribute A and each value is taken as a tuple.
- 7) (Constraint) Given DGQL query expression e , then $\xi[C](e)$ is a DGQL query expressions, where C a condition as found in selection; The schema of the expression is that of e ;

Definition 1: A DGQL query is an assignment sequence of the form

$$R_1 = e_1; \dots; R_{n-1} = e_{n-1}; \omega(e_n),$$

where each e_i is a DGQL expression and ω is either \min or \max . That is, a DGQL query is a sequence of assignments followed by an optimization operator.

A *valid* DGQL query is one that always uses a relation name that is either a base relation name, or one that has been defined (i.e., on the left-hand-side of an assignment) earlier in the sequence. We henceforth assume we only deal with valid DGQL expressions if validity is not mentioned.

The example in Section IV can be easily rewritten in the DGQL algebraic query form, with the each view being an assignment followed by `MINIMIZE total.cost`.

We now turn to define the semantics of DGQL queries. For simplicity, we can assume that in a DGQL query, each e_i is either a single relation name (base relation name or one of R_1, \dots, R_{i-1}) or a single operator applied to operands in the form of single relation names, and e_n must be one of R_i . In other words, each assignment is a “simple step” of copying a relation or applying one operation over the previously defined relations. We also assume the optimization operator only applies to single relation name.

Definition 2: A DGQL query is a *basic assignment sequence* if each assignment is a simple step as described above.

Since a DGQL query can be rewritten into a basic assignment sequence by introducing more assignments, without loss of generality, we *henceforth assume all DGQL queries are basic assignment sequences* if not mentioned otherwise.

Intuitively, (1) the assignments of a DGQL query is to give all “feasible” instantiations of all the relations for the left-hand-side of each assignment such that the “input” and “output” of each assignment is “correct”, i.e., given the input relations, the left-hand-side relation is actually the result of the operator applied to the given operands; (2) the optimization operator as the last step of a DQQL is to find one such feasible instantiation such that the e_n value is the minimum or maximum (depending on either \min or \max is used in the last step) among all the feasible instantiations. For example, the DGQL query in Section IV obtains an assignment to all the views, including the view `orders` such that the total cost is minimized. In this way, the optimization done by DGQL automatically fills the `orders` table for the emergency response application.

Different from the standard SQL, where the operands and the operator determine the output relation, the DGQL operator augmentation introduces non-determinism. That is, for the augmentation operator, infinitely many “output” relations may correspond to an “input” relation. Therefore, a DGQL query can have many feasible instantiations for the assignment sequence. And the optimization operator in the end is to choose one feasible instantiation.

Formally, consider a DGQL query $R_1 = e_1; \dots; R_{n-1} = e_{n-1}; \omega(e_n)$. A (non-deterministic) *execution path* is a sequence (r_1, \dots, r_{n-1}) of relations over the schemas of e_1, \dots, e_{n-1} , respectively. A *partial execution path* is a prefix of an execution path, i.e., a sequence of relations (r_1, \dots, r_i) , $1 \leq i < n$, of relations over the schemas of e_1, \dots, e_i respectively, or the empty prefix ϵ .

We define the *feasibility* of a partial execution path recursively as follows. The empty execution path ϵ is *feasible*. Recursively, (r_1, \dots, r_i) , $1 \leq i < n$, is feasible if (1) r_1, \dots, r_{i-1} is feasible, and (2) the following is satisfied:

- if e_i is a *traditional* or *aggregation* operation, then r_i is the result of e_i computation when operands from R_1, \dots, R_{i-1} are replaced with relations r_1, \dots, r_{i-1} respectively
- if e_i is an augmentation operation $\alpha[A_1, \dots, A_m \in R_j](R_k)$, where $1 \leq k, j < i$, r_i has the schema of R_k extended with A_1, \dots, A_m , and is constructed by joining a tuple from r_k with a single tuple (non-deterministically chosen) from $\pi_{A_1, \dots, A_m}(r_j)$. Similarly, if e_i is $\alpha[A \in domain](R_k)$, where $1 \leq j < i$, r_i has the schemas of R_k extended with A , and is constructed by joining a tuple from r_k with a single value (non-deterministically chosen) from the *domain* of A .
- if e_i is a constraint operation $\xi[C](R_j)$, $1 \leq j < i$, the constraint C is satisfied by r_j and $r_i = r_j$.

Definition 3: The *semantics* of a DGQL query

$$R_1 = e_1; \dots; R_{n-1} = e_{n-1}; \omega(e_n)$$

is defined as the function

$$\Phi : \mathcal{D}_1 \rightarrow 2^{\mathcal{D}_2},$$

where \mathcal{D}_1 is the set of all relational databases over the base database schemas, and \mathcal{D}_2 is the set of all databases over

the schema S_1, \dots, S_{n-1} of the expressions e_1, \dots, e_{n-1} , respectively, as follows. For a given database d , $\Phi(d)$ is the set of all databases $d' = \{r_1, \dots, r_{n-1}\}$ such that it gives the optimal value among all the feasible execution paths, i.e., $\Phi(d) = \{(r_1, \dots, r_{n-1}) \mid (r_1, \dots, r_{n-1}) \text{ is a feasible execution path and the } e_n \text{ value using } (r_1, \dots, r_{n-1}) \text{ is min/max among all feasible execution paths}\}$

Note that the optimal value may not lead to a unique feasible execution path, i.e., more than one assignment to relations r_1, \dots, r_{n-1} may lead to the same optimal value for e_n . We say that a database $d' = \{r_1, \dots, r_{n-1}\} \in \Phi(d)$ is a non-deterministic answer to the query. If $\Phi(d) = \emptyset$, we say that \perp (to denote infeasibility) is returned and assigned to R_1, \dots, R_{n-1} .

VI. EVALUATING DGQL WITH MP

In this section, we present a reduction procedure to translate DGQL queries to mathematical programming (MP) formulations. We first introduce MP with some discussion, and then give a *sound* and *complete* reduction procedure.

A. MP and its Computational Complexity

A mathematical programming (MP) problem has the form “ $\min f(\hat{x})$ s.t. $C(\hat{x})$ ” where \hat{x} is a vector of variables that range over domain \hat{D} , f is an *objective* function from \hat{D} to the set of reals R , and $C(\hat{x})$ is a constraint. The constraint $C(\hat{x})$ is associated with a Boolean function that, given an instantiation of vector $\hat{a} \in \hat{D}$ to \hat{x} , evaluates to True or False. A solution to the optimization problem is a vector \hat{a} in \hat{D} that satisfies the constraint C so that $f(\hat{a}) \leq f(\hat{b})$ for all vectors \hat{b} that satisfy C . Note that *min* can be replaced with *max* in this formulation without difficulties.

Formal optimization models are solved using MP algorithms. MP algorithms heavily depend on the type of the problem, determined by the domains of the variables (e.g., reals, integers, binary, finite domain), the structure of constraints C (e.g., linear, quadratic, polynomial, etc.), and the form of the objective function f . As an example, Linear Programming (LP) is a class of MP where variables range over reals, constraints are inequalities between linear arithmetic expressions, and the objective function is linear.

The LP problem was proven to be in \mathcal{P} in [8]. However, the class of SIMPLEX algorithms [4] is still the most widely used in practice, although its worst case complexity is exponential in the dimension (i.e., the number of variables). On the other hand, even the $\{0, 1\}$ Integer Linear Programming (ILP), is known to be NP-complete [10]. The Branch and Bound algorithm is a popular algorithm for ILP and MILP (mixed integer LP). While exponential in the worst case, it has the nice property of reaching intermediate (not necessarily optimal) solutions. In this paper, we use MILP to encode DGQL queries and derive solutions.

B. Reduction Procedure

The key idea of the reduction is to use a symbolic table formation to represent each step of a DGQL query. For this

purpose, we assume that every tuple in a relation has a globally unique tuple identifier *tid* and when new tuples are generated, new *tid*'s will be generated correspondingly.

Definition 4: A *symbolic table* is a standard relational table with the following additions: (1) an attribute can be of a constraint variable name type (or CVN type), and the domain of a CVN attribute is variable names, (2) there must be a special attribute of CVN type named TAF, for tuple-active-flag, with domain of binary variable names of the form $\text{taf}[\text{tid}]$, where *tid* is the tuple id of the tuple.

As an example, below is a symbolic orders table:

TAF	sup	item	erloc	qty
taf[1]	s1	i1	l1	qty[1]
taf[2]	s2	i1	l1	qty[2]

where TAF and qty are CVN type attributes. The idea of TAF attribute is to indicate if the particular tuple exists in the table in a feasible execution path, and the CVN attributes allow possible instantiations.

Symbolic tables are to be instantiated as regular relational tables by instantiation of variables. More specifically, given an instantiation of variables appearing in a symbolic table, the symbolic table becomes a regular table, minus the TAF column as follows: For each tuple t in the symbolic table, if $\text{taf}[\text{id}]$ is instantiated to 1, then the resulting regular table contains the tuple t with other CVN attributes instantiated and TAF attribute dropped. For example, in the above table, if $\text{taf}[1]=1$, $\text{taf}[2]=0$, $\text{qty}[1]=20$, and $\text{qty}[2]=4$, then the resulting table is:

sup	item	erloc	qty
s1	i1	l1	20

Note that the TAF attribute is dropped.

The reduction to MILP is as follows: given a DGQL query $R_1 = e_1; \dots; R_{n-1} = e_{n-1}; \omega(e_n)$, we generate a symbolic table for each R_i . But in order to represent all the feasible execution paths, constraints are to be added so all and only feasible execution paths are represented. In other words, the *reduction result* will be a triple

$$(\{s_1, \dots, s_{n-1}\}, V, C),$$

where $\{s_1, \dots, s_{n-1}\}$ are symbolic relational tables, V is the set of all constraint variables, including those that appear in $\{s_1, \dots, s_n\}$, and C is a set of constraints over V . The schema of each s_i , $1 \leq i \leq n$, will be the schema of e_i extended with the TAF attribute.

Definition 5: Given an instantiation all the variables in V that satisfy C , the reduction result $(\{s_1, \dots, s_{n-1}\}, V, C)$ is instantiated to (r_1, \dots, r_{n-1}) such that each r_i is the corresponding instantiation of s_i using the variable instantiation (and the TAF attribute dropped).

Before we discuss the procedure to obtain a reduction from a DGQL query, we formalize the correctness of the reduction result. As discussed above, the symbolic tables together with the constraints should represent all and only feasible execution paths. To formalize this idea, we have:

Definition 6: The reduction result $(\{s_1, \dots, s_{n-1}\}, V, C)$ of a DGQL query $R_1 = e_1; \dots; R_{n-1} = e_{n-1}; \omega(e_n)$ is said to be *correct* if

Soundness:

Every instantiation (r_1, \dots, r_{n-1}) of (s_1, \dots, s_{n-1}) that satisfies constraint C is indeed a feasible execution path of the DGQL query; and

Completeness:

Every feasible execution path is an instantiation of (s_1, \dots, s_{n-1}) that satisfies C .

The reduction steps follow the traditional relational algebra approach in the sense that each step we just generate a new symbolic table and add appropriate constraints. These steps are summarized in Algorithm 1.

Algorithm 1 Correct reduction

Input: A DGQL query $R_1 = e_1; \dots; R_{n-1} = e_{n-1}; \omega(e_n)$

Output: A correct reduction $(\{s_1, \dots, s_{n-1}\}, V, C)$

Method:

- 1: Let $V = \emptyset$ and $C = True$.
 - 2: **for** each base table B **do**
 - 3: generate a symbolic table by adding a TAF attribute
 - 4: for each tuple in B with tuple id tid , use CVN $\text{taf}[tid]$ as the value under TAF, and let $C = C \wedge \text{taf}[tid]=1$. Add $\text{taf}[tid]$ to V .
 - 5: **end for**
 - 6: **for** $i = 1$ to $n - 1$ **do**
 - 7: Generate symbolic table s_i from the operator of e_i and the input symbolic tables used in e_i (these input symbolic tables must be from s_1, \dots, s_{i-1} and those corresponding to the base tables).
 - 8: Generate the corresponding C_i
 - 9: Add the new variables used in s_i and C_i to V
 - 10: Let $C = C \wedge C_i$
 - 11: **end for**
 - 12: **Return** $(\{s_1, \dots, s_{n-1}\}, V, C)$.
-

Algorithm 1 produces symbolic tables s_i recursively using the symbolic tables generated earlier. The base case (steps 2-5) is to make the base database tables symbolic by adding a TAF attribute and the corresponding variables and constraints, indicating that all the tuples are “active” in the base tables. At each step of the reduction (steps 7-10), a new symbolic table for e_i is generated and constraints and variables added to C and V , respectively. We now describe the key idea behind these steps.

Step 7 of Algorithm 1 is to generate a symbolic table for s_i given one or two symbolic tables as the input. The schema of the output table is easily found by using the formal definition of the DGQL and traditional relational algebra operators. Setting up all the tuples in s_i is easy for most operators, but a bit challenging for some. Intuitively, we want to set up all the possible output tuples in s_i and use constraints involving $\text{taf}[tid]$ to indicate if the tuple should actually exist in a feasible execution path. For example, if the operator is the selection operation, then the output table can be simply a copy of the input table, but in the constraint we code that an output tuple exists only if the selection condition is true. The reason we need to use constraint instead of simply dropping

the tuple is that the selection condition may be applied to a CVN entry whose value is unknown at the reduction time.

The number of all possible tuples in s_i can be exponential in some cases. For example, when “group by” is done on a CVN attribute, we need to set up s_i to prepare for all possible groupings since we do not yet know the values of these CVN attributes. In practice, we may want to restrict “group by” only on non-CVN attributes so that the grouping will be known at the reduction time and each group will correspond to one possible tuple in s_i (although the actual values in the tuple may remain CVN and unknown at the reduction time).

Once the symbolic output table is setup, the remaining task is to generate the constraint (step 8). This step is basically to encode the correspondence between the input tuples and the possible output tuples. Since we know which symbolic tuple in the output is generated by which input tuples, this task is relatively easy as this is in reality a tuple level task.

We are now ready to address the correctness of the reduction. We first characterize the correct reduction steps 7-8, and then summarize the correctness of Algorithm 1.

Definition 7: Given two symbolic tables s_1 and s_2 and a DGQL operator $o(s_1, s_2)$, a symbolic table s_3 and constraint C is a *correct encoding* of $o(s_1, s_2)$ if, assuming all variables are in V , for all instantiations I of V , we have $I(s_3) = o(I(s_1), I(s_2))$ if and only if $I(V)$ satisfies C . Here, the semantics of o is defined in the formal semantics of DGQL in Section V, and $I(s_j)$ is the table obtained by (1) instantiating all the variables appearing in s_j via I , and (2) dropping all the tuples such that TAF entry is 0. The correctness of one operand operation is defined similarly.

The above definition is our guideline for detailed encoding of each operator. In fact, we can prove that each reduction step is correct by the above definition. Now we can state:

Theorem 1: If Algorithm 1 correctly encodes each operator in steps 7-8, then the reduction produced by Algorithm 1 is correct, i.e., both sound and complete.

Once a correct reduction result is obtained, the result of a DGQL query can be derived as indicated in Algorithm 2. Basically, we solve the corresponding MP problem given by the correct reduction result (step 1). If the constraint is infeasible, then we return \perp (steps 2-3). Otherwise, we use the instantiation of the variables found in the MP solution to instantiate all the symbolic tables (steps 5-6).

Theorem 2: Algorithm 2 correctly, by Definition 3, computes the DGQL query result.

VII. EXPERIMENTAL EVALUATION

The running time of mathematical programs are 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 DGQL (and the reduction process as described in Section VI-B) is that the extra views and tables used in modeling add substantial overhead by introducing redundant or unnecessary variables

Algorithm 2 Finding query result from correct reduction

Input: A correct reduction $(\{s_1, \dots, s_{n-1}\}, V, C)$ of the DGQL query $R_1 = e_1; \dots; R_{n-1} = e_{n-1}; \omega(e_n)$

Output: Query result

Method:

- 1: Solve the MP problem “ $\min/\max(e_n)$ s.t. C ” and if the problem is feasible, generate an instantiation of the variables in V to yields the optimal solution to the MILP problem.
- 2: **if** constraint C is infeasible **then**
- 3: return \perp
- 4: **else**
- 5: Instantiate the symbolic tables s_1, \dots, s_{n-1} with the variable instantiation found in step 1.
- 6: Return these instantiated tables.
- 7: **end if**

and constraints. It is possible a simpler or more concise modeling may lead to a significantly faster solution.

Our hypothesis is that problems described using DGQL are solved as efficiently as when described using any other modeling tool. To test this, we took the ER problem from Section II and manually created a concise AMPL formulation of the same problem. We then generated several database instances of varying problem size and compared the running time when solved using this AMPL model and through DGQL reduction. The AMPL implementation of can be found below.

```

set Vendors;
set Items;
set Locations;
param PricePerUnit{Vendors,Items}>=0;
param Available{Vendors,Items}>=0;
param Requested{Locations,Items}>=0;
var Orders{Vendors,Items,Locations}>=0;
var OrderPlaced{Vendors,Items,Locations} binary;
var ItemsPurchased{Vendors,Locations}>=0;
var VendorShipped{Vendors,Locations} binary;

minimize total_cost:
    sum{v in Vendors} sum{i in Items} sum{l in Locations}
        PricePerUnit[v,i]*Orders[v,i,l];

subject to
available_vs_purchased {v in Vendors,i in Items}:
    Available[v,i] >= sum{l in Locations} Orders[v,i,l];
requested_vs_delivered {l in Locations,i in Items}:
    Requested[l,i] <= sum{v in Vendors} Orders[v,i,l];
order_placed {v in Vendors,i in Items,l in Locations}:
    OrderPlaced[v,i,l] = 0 ==>
        Orders[v,i,l] = 0 else Orders[v,i,l] >= 1;
items_purchased {v in Vendors,l in Locations}:
    ItemsPurchased[v,l]=sum{i in Items} OrderPlaced[v,i,l];
vendor_shipped {v in Vendors,l in Locations}:
    VendorShipped[v,l] = 0 ==>
        ItemsPurchased[v,l]=0 else ItemsPurchased[v,l]>=1;
vendor_restriction {l in Locations}:
    sum{v in Vendors} VendorShipped[v,l] <= 3;
    
```

The DGQL query and the AMPL program were run using ILOG CPLEX 12.1 as the solver. The test was on Windows XP running on a 2.0GHz Core 2 Duo with 4GB of RAM. In both cases, the tuning parameters passed to ILOG CPLEX were the same and were typical for solving MILP problems.

The size of the ER problem here is parameterized by the number of vendors, items, and ER locations. Given these parameters, a problem is generated by randomly instantiating the database tables supply and demand. Note that problem size corresponds to the number of decision variables in

orders as well as the indicator variables (tuple-active-flags) implicit in vendor_restrictions. Thus, we compute this directly from the problem parameters, i.e., $decision\ variables = vendors * items * erlocs$ and $indicator\ variables = vendors * items * erlocs + vendors * erlocs$.

Each random problem generated was solved using both the DGQL reduction and the AMPL program, and the solution time was measured. This is the CPU time spent searching for an optimal solution and does not include the time to load the program off disk, or in the case of DGQL from the database. Only random databases that contained a feasible solution were included in the results. The solution time for both the DGQL and AMPL implementations was less than 10ms for every randomly generated infeasible problem. The results of this experiment can be seen in Figure 1.

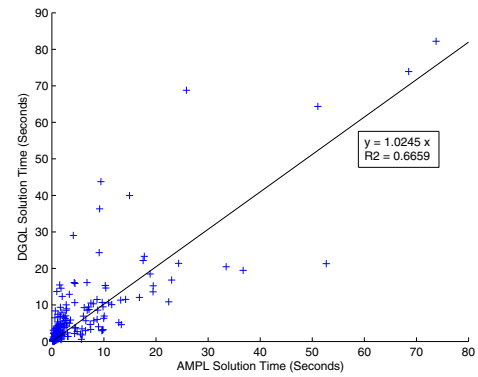


Fig. 1. Scatter plot of solution time

Note in the graph, a linear regression through the origin has slope 1.0245. This implies the AMPL model is on average $\approx 2.45\%$ faster than DGQL. This difference is small and not significant when considering the large variation in solution time between the two systems for the same problem.

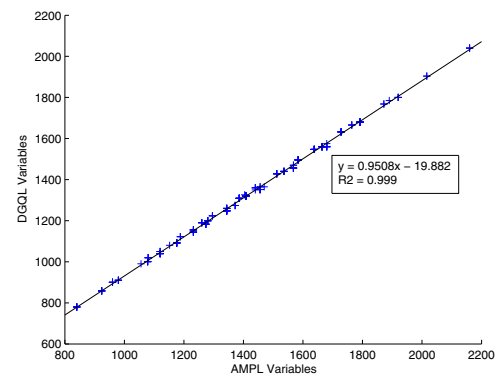


Fig. 2. Scatter plot of total number of variables

On closer inspection, it appears the advantage AMPL has is limited only to those problem instances where a tight solution is found by CPLEX quickly. Our DGQL implementation uses Java and JNI to communicate with CPLEX and this may add overhead to small problems. Moreover AMPL performs limited presolving when converting problems into a special format CPLEX understands which may reduce CPLEX solution time.

Additionally, DGQL appears to do better on problem instances where a larger parameter space must be searched. This is due in part to the fact that the AMPL formulation has an extra *vendors*erlocs* number of variables in its solution space (see Figure 2). As the ER problems become larger, this extra number of variables will have a measurable impact on solution time. It will be interesting to investigate further if these extra variables are a limitation of the AMPL modeling language or just our formulation of this problem.

VIII. CONCLUSION

In this paper, we have introduced a database language for decision optimization. We have defined its formal syntax and semantics, and provided a canonical implementation of the queries using mathematical programming. We have used examples to show that the overhead of using a high-level language is small when compared with manually crafted mathematical programming formulation. With added benefit of intuitive high-language, we believe that DGQL provides a practical solution for decision optimization, especially when an underlying database is involved and database applications already exist.

As a database language, DGQL opens up many research questions. For example, what is the expressiveness of DGQL: how to characterize it, and if there is a completeness as in relational algebra. Optimization opportunities are abundant and we only touched upon a few during our canonical

reduction process. Standard database techniques may help, but more interesting is how to use the high-level nature of the language to help finding optimal solutions, as it is well known in database community that more semantic information usually translates to better query optimization.

REFERENCES

- [1] AMPL. <http://www.ampl.com/>.
- [2] R. F. Boisvert, S. E. Howe, and D. K. Kahaner. Gams: a framework for the management of scientific software. *ACM Trans. Math. Softw.*, 11(4):313–355, 1985.
- [3] A. Brodsky and H. Nash. CoJava: Optimization modeling by nondeterministic simulation. In F. Benhamou, editor, *Proceedings of Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2006.
- [4] Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, N.J., 1963.
- [5] P. Fritzson and V. Engelson. Modelica - a unified object-oriented language for system modelling and simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90, London, UK, 1998. Springer-Verlag.
- [6] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson Education (US), 2international ed edition, August 2008.
- [7] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
- [8] L. Kachiyan. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–4, 1979.
- [9] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. on Database Sys.*, 12(4):566, Dec. 1987.
- [10] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, June 1998.