# MOSDEX: Proposal for a New Standard for Data Exchange with Optimization Solvers

Dr. Jeremy A. Bloom
Retired
jeremyblmca@gmail.com

Dr. Alan King
IBM

Professor Matthew Saltzman
Clemson University

August 11, 2022

**Abstract**

This paper proposes a new standard, called MOSDEX (Mathematical Optimization Solver Data EXchange), for managing the interaction of data with solvers for mathematical optimization. The rationale for this standard is to take advantage of modern software tools that can efficiently handle very large datasets that have become the norm for data analytics in the past few years. MOSDEX is based on several principles: independence from and support for multiple optimization solvers and multiple algebraic modeling languages, separation of model and data, relational data modeling, and incorporation of standard optimization modeling artifacts. MOSDEX uses the widely adopted JSON data format standard to take advantage of JSON support in a variety of programming languages including Java, C++, Python, and Julia. The paper demonstrates the principles of MOSDEX through examples taken from a well-known optimization problem. A companion provides a fuller description of the MOSDEX syntax and a library of examples of MOSDEX representations of optimization problems.

## 1. Introduction and Basic Rationale

This paper proposes a new standard, called MOSDEX (Mathematical Optimization Solver Data EXchange), for managing the interaction of data with solvers for mathematical optimization. The rationale for this standard is to take advantage of modern software tools that can efficiently handle very large datasets that have become the norm for data analytics in the past few years. Much of the voluminous literature on mathematical optimization, even in recent times, has focused almost entirely on the efficiency of solution algorithms, to the neglect of the computational effort required to prepare the data for those algorithms and to retrieve solutions from a solver. Yet, in our experience, the effort, both intellectual and computational, required for the latter two activities often dominates the effort required for the algorithms themselves. The standard we propose aims to remedy that deficiency.

Our thinking is guided by the view that solving an optimization problem is but one step in a sequence of transformations of data from its origins in an enterprise data store to its ultimate destination in operational systems that implement enterprise decisions. The proposed MOSDEX standard intends to support this entire stack. The key to this endeavor lies in three principles:

1. Treat data as a stream in which, to the greatest extent possible, transformations occur without intermediate realizations in memory;

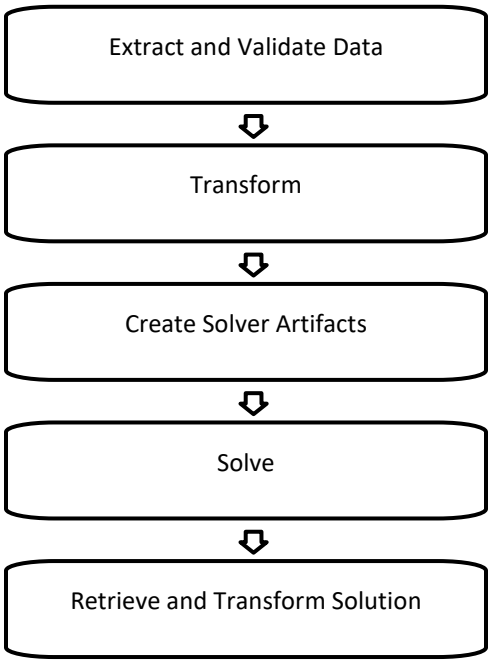
 

2. Use a data manipulation language (i.e. Structured Query Language) to implement and document the transformations that take place;
3. To the greatest extent possible, provide a standard that is implementable and neutral with respect to the host programming language, the optimization solver, and the data management system employed.

Prior work, summarized in the next section, addresses several different approaches for specifying data for optimization. At one end of a spectrum, *instance* formats, which include the venerable MPS as well as LP, OSiL, and many others, provide specific data for a solver's representation of an optimization problem; frequently, the instance data maps directly to the solver's internal data structures. At the other end, *algebraic modeling languages (AML)*, such as AMPL, GAMS, and OPL, provide a mathematical representation of an optimization model and, typically, use a separate representation of the data for a particular instance; the language processor transforms the model and data to populate a solver's internal data structures. While MOSDEX intends to supersede existing instance formats, we believe it is complementary to AMLs, providing standardization of the data representation within the mathematical syntax for representing a model.

The chief issue with instance formats is that they represent the end product of a series of transformations upstream of the solver. These transformations are often ad hoc, not standardized, and often they are undocumented. In essence, most applications that use optimization are custom designed, and a great deal of effort is needed to create the data manipulations upstream and downstream of the solver (see figure 1). MOSDEX is designed to support this entire architecture, using data transformation tools based on SQL.

*Figure 1: Data Flow in an Optimization Application*



## 2. Prior Work

While mathematical optimization has, of course, a voluminous literature stretching back to its invention by George Dantzig, spanning both algorithms for solving optimization problems and formulation of optimization models, work on preparing data for optimization is somewhat sparse. What prior work on this topic exists falls into four categories: instance formats, structure conveying formats, algebraic modeling languages, and use of SQL in optimization. We survey each of these categories below.

An *instance* of an optimization model is a representation of the data that maps directly to the model's variables, constraints, objectives, and terms (collectively, the modeling *artifacts*) that a solver uses in computing a solution (Gassmann, Ma, & Martin, 2011). The most widely used instance format is MPS, developed in the 1970s by IBM, which "has emerged as a de facto standard ASCII medium among most of the commercial LP solvers." (Wikipedia, MPS format, 2018). MPS brings several advantages that, we believe, should be preserved when considering an alternative:

1. **Sparsity**: MPS supports a sparse data format that requires specifying only non-zero elements of the data. Since most real-world optimization problems are highly sparse (1-10% non-zeros), this feature results in substantial reduction of data volume and computational effort. In addition, all professional-grade solvers use sparse data structures internally.
2. **Text-based**: MPS files are ordinary text files and are reasonably easy for humans to read.
3. **Non-proprietary**: Although MPS began is a proprietary standard for an IBM solver, it became widely adopted, and today almost all solvers support it.

However, despite these advantages, MPS and other instance-based formats suffer from a number of shortcomings. Among these, MPS's fixed-column format, while archaic, is perhaps the least important, since that aspect is rather easily overcome and many solvers that accept MPS files do not enforce it. More important are the following:

a. **Lack of an output standard**: MPS is a format for input and there exists no corresponding standard for output from an optimization solver. Some other instance formats do specify output standards, but many solvers use unique output formats.
b. **Lack of model/data separation**: This aspect means that the model's artifacts are intertwined with the data that populates them rather than specified independently of the data comprising any particular instance. It is considered a best practice to separate them, for a number of reasons discussed below. Furthermore, the widely used algebraic modeling languages are designed for model/data separation.
c. **Difficulty in scaling:** One reason for specifying the model and data separately is that it allows for scalability – the model remains unchanged as the size of the data changes. In practice, an optimization application often consists of a family of related instances in which a model, representing, for example, a distribution network, does not change but the data represents different numbers of entities, such as warehouses, stores, and routes between them.
d. **Unable to represent structure**: Instance formats generally are unable to elucidate the higher level structure of an optimization problem. We elaborate in Section 6.
e. **Lack of indexing**: One way to achieve scalability and convey structure is to use indexing to represent groups of related variables or constraints, a common practice in mathematics. In MPS and other instance formats, each variable is represented as a column and thus there is

a column record for each. With indexing, each family of variables could be represented by a single artifact (although the use of columns to specify matrix data would also need to be revised as discussed in the next item).

f. **Column orientation**: In MPS, the matrix elements (i.e. the coefficients of each variable in the constraints) are specified in the column records. (This aspect is probably an historical artifact of the way that a sparse matrix was specified in the original IBM solver.) LP is a row-oriented format; other instance formats (notably OSiL) can use either orientation. While column orientation is appropriate for some optimization models (for example a model constructed using Dantzig-Wolfe decomposition), for others a row orientation is more appropriate. Indeed, in mathematical notation, an optimization model is usually represented by its constraints, and most of the widely adopted modeling languages favor specification by constraint (row orientation) over specification by variable (column orientation).

g. **Extensions beyond linear models**: Originally, MPS was intended to represent purely linear optimization problems. As solvers began to support integer and mixed-integer problems, their developers extended the MPS format to accommodate them, although no standard for such extensions emerged. Furthermore, solvers began to exploit special model structures in their algorithms, such as special-ordered sets or indicator constraints, again extending the MPS format to represent them. A modern data exchange standard needs to support these extensions, as well as quadratic, conic, and other nonlinear formulations, in a standard way.

A more modern alternative to MPS is called *Optimization Services instance Language (OSiL)*. OSiL is part of an overall architecture called *Optimization Services (OS)* (Fourer, Ma, & Martin, 2010), which focuses on implementing optimization solving as a service over the internet. OS defines standards for the activities necessary to support decentralized optimization – representation of optimization instances, results, and solver options; communication between clients and solvers; and discovery and registration of related software. OS uses a widely adopted data format language, XML, and enables use in a multiprocessor computing environment by using Simple Object Access Protocol (SOAP). However, given the complexity of XML and its consequent difficulty for human readers, we have chosen to implement MOSDEX in JSON, which has a much simpler syntax and which has been widely adopted for similar applications.

OSiL (Fourer, R., Ma, J., and Martin, K. , 2010), a component of OS, is an XML-based language for representing instances of optimization problems including linear programs, mixed integer programs, quadratic programs, and nonlinear programs. While OSiL represents a significant update to the MPS standard, it nevertheless has significant limitations that it shares with other instance formats, as discussed above. Because an instance format maps directly into a solver's corresponding in-memory representation of a problem instance, it also does not convey the higher-level structure that exists in many large optimization models, notably stochastic programs, that can be exploited in specialized solver algorithms.

A major problem with instance formats generally is that they represent the end product of a series of transformations upstream of the solver; in effect, they implement the "Create Solver Artifacts" transformation in Figure 1, without addressing the other upstream and downstream transformations. These other transformations are often ad hoc, not standardized, and often they are not well documented. This is the gap that MOSDEX intends to fill.

Addressing the desirability of conveying structure, Colombo et al. (2009) develop an object-oriented format that breaks an optimization model into parametrized blocks of sub-models, each of which is identified to a solver, which enables applying specialized algorithms. Their approach is implemented as a combination of pre-and post-processing phases of an algebraic modeling language, AMPL. Unlike traditional modeling languages, their approach does not scramble the block structure of the model. While not strictly a data exchange format, the ideas developed are not limited to a particular modeling language and can be adapted more generally (see Section 8 below).

Algebraic modeling languages, among them AMPL (AMPL Optimization Inc.), GAMS (GAMS Development Corp.), and OPL (IBM Corporation), have represented a significant advance in optimization modeling, enabling creating models as natural mathematical constructs familiar to most operations research developers. In a sense, they provide a layer of syntax above a data exchange format that includes both mathematical operators, such as summation, and sophisticated methods for manipulating index sets, such as slicing, which, in our opinion, are essential to conveying structure in optimization models. There are many ideas embodied in algebraic languages that can be usefully employed in a data exchange format as well, especially the notion of model/data separation. Furthermore, to the extent that one practices that kind of separation, a data exchange standard can usefully be adopted as the data format in an algebraic language. Thus the two approaches are complementary.

Finally, there are a few papers that address the relationship between optimization modeling and data manipulation languages, most notably SQL. Several authors have, apparently, independently observed the strong relationship between these two domains. MOSDEX itself was inspired by previous work by one of the authors (Bloom, 2017). Fourer (1997) discussed the fundamental principles of database construction for large-scale mathematical programming, using a steel mill planning model as an example. In particular, he demonstrated how different formulations of the optimization model generated different database structures. Fourer's paper uses relational algebra to describe the tables that underlie the model formulation. A little later, Atamturk et al. (2000) discussed optimization modeling based on relational algebra, in which modeling related activities, such as model formulation, model instantiation, and model and instance management, are done using relational operations such as selection, projection, and predicated join. Atamturk's paper explicitly formulates SQL queries that create the tables that underlie the example model formulation, a production and distribution example. Periodically, the same insights have been rediscovered by others. Despite this history, however, the relationship between modeling and databases has not become part of the main stream in optimization modeling. This relationship derives from the fundamental role index sets play in optimization models (see Section 6). However, to our knowledge, these prior works seem to have been one-off efforts that did not produce generally applicable standards or software. Perhaps the separation reflects a long-standing cultural difference between mathematical modeling and data engineering. However, we believe that, with modern database technology and the increasing use of application architectures linking data to optimization in an integrated stack, the time has come to take advantage of this synergy.

## 3. MOSDEX Design Principles

Below, we will explain in detail MOSDEX, the proposed new standard. However, here is a brief summary our design principles:

a. **Efficient for machines, readable by humans**: From an architectural perspective, the role of a model is to move data from its source in enterprise systems into (and out of) the solver's internal data structures. Since many optimization applications are highly automated, data must flow with as little processing overhead as possible. However, since humans design, implement, and use these applications, it is essential that they be able to read easily the content of data files. People need to verify the model design and to correct any bugs, such as misaligned indices and malformed row and column labels. Instance formats generally do not facilitate such tasks. Even LP format, arguably one of the most user friendly, becomes very difficult to read with more than a handful of variables and constraints. Furthermore, the ability to see the larger structure of the model contributes significantly to readability, and instance formats generally do not show such structure. Thus, MOSDEX design also has a primary goal to support readability by humans.
b. **Represent the data in relational form**: Use the well-known structure of relational data bases, that is, a set of 2-dimensional tables each consisting of a fixed column schema and an indeterminate number of rows.
c. **Use the JSON (JavaScript Object Notation) standard**: Tie the standard format for optimization problems to a widely used data format standard to take advantage of support for the standard in various programming languages. We discussed above the reasons for choosing JSON over XML, the other widely adopted data format standard.
d. **When necessary, augment the data representation with mathematical modeling artifacts in the new standard**: When using a modeling language, the data representation should be sufficient to fully specify an optimization problem. However, a full replacement of MPS would also need to specify the modeling artifacts. The MOSDEX standard supports linear, integer, and mixed-integer linear models. It is also possible to extend MOSDEX to accommodate quadratic formulations and various special structures (see Section 8). However, MOSDEX currently does not support general nonlinear formulations, which increase complexity enormously due to the need to represent more general mathematical expressions. (Note however, that the relational data representation would also support nonlinear formulations.)
e. **Extensibility**: Enable extension of MOSDEX to accommodate modeling designs beyond linear and quadratic models, in particular decomposition, stochastic and general non-linear models (see Section 8). To the extent possible, make extensions systematic and parsimonious, based on a few core structures, and avoid proliferation of ad hoc artifacts. We believe that use of JSON and SQL facilitates extension of MOSDEX.

## 4. MOSDEX Overview

At the highest level, a *MOSDEX File* is a collection of *Modules*, which represent either data only or an optimization problem (modeling artifacts with or without with data). Each Module in turn is a collection of *Tables*. A Table represents a table in a relational database. Subclasses of Table represent Data and modeling artifacts, such as Variable and Constraint. Every Table has a *Schema*, which defines its fields and their data types. Data tables can have any reasonable schema, while the schemas of the modeling artifacts are largely fixed by the requirements of an optimization solver. A Table can have either *instance* or *query* form. Instance form tables contain data while query form tables use SQL queries to reshape and populate data from other tables. A MODEX Module can contain both instance- and query-form

tables, although using query-form tables presupposes the presence of a database in the software stack that executes the optimization application.

In order to illustrate specification of a simple linear program in MOSDEX, consider Equation 1, a transshipment network borrowed from AMPL (Fourer, Gay, & Kernigan, 2003).

*Equation 1: The Transshipment Model*

$$\textbf{Minimize } totalCost = \sum_{(i,j)\ in\ routes} cost[i,j] * ship[i,j]$$

$$\textbf{Subject to}:$$

$$\textbf{for all } \{k\ in\ cities\}$$

$$balance[k]: \sum_{(k,j)\ in\ routes} ship[k,j] - \sum_{(i,k)\ in\ routes} ship[i,k] = supply[k] - demand[k]$$

$$0 \leq ship[i,j] \leq capacity[i,j] \textbf{ for all } (i,j)\ in\ routes$$

This example will develop two MOSDEX representations. One uses purely instance-form tables, illustrative of how MOSDEX compares with MPS and other instance formats, while the other uses query-form tables, illustrative of the advantages of SQL in this context. To be clear, MOSDEX can be used in either form. A few general comments are in order before we delve into the details of this example. First, MOSDEX is a derivative of JavaScript Object Notation (JSON), and therefore, MOSDEX files adhere to the JSON standard (see (JSON)). Second, MOSDEX is specified using a standard JSON Schema (see (JSON Schema))[i]. Among other uses, the MOSDEX schema enables validating a MOSDEX file to assure conformance with the MOSDEX standard. Third, the example has been laid out visually to facilitate clarity for human readers; however, the visual layout is not part of the standard, and generally JSON does not enforce any particular layout (although many parsers offer a "pretty print" option). Certain elements of the MOSDEX standard, such as the HEADING objects, are present specifically to provide information for human readers and are not processed by a MOSDEX parser; such elements are generally optional. Since the JSON standard does not allow comments (although many JSON parsers support C-style comments), these MOSDEX elements provide an alternative means to annotate a file.

Finally, JSON has three fundamental elements: *objects, arrays,* and *primitives*. A JSON object (similar to a Python *dictionary* or a Java *map*) is an unordered list of *key* : *element* pairs, or *members*, where each key, or *field name*, is a string; it is enclosed within curly braces, { and }, and a member's key and element are separated by a colon, with the object's members separated by commas. Keys must be unique within an object. In MOSDEX, the field name is always a *keyword*, denoted by using all capital letters.

A JSON array (similar to an *array* or *list* in Python or Java) is an ordered list of elements; it is enclosed in square brackets, [ and ], and the elements are separated by commas. Array elements may be of mixed types. Both objects and arrays can be nested within each other.

Finally, JSON supports the following primitive types: *strings* of Unicode characters enclosed in double quotes, decimal *integers*, decimal floating point numbers with or without an exponent (*double* in Java), and null. (*boolean*, another JSON primitive, is not used in MOSDEX.) Additionally, MOSDEX allows another number type, *IEEE Doubles*. These are represented as strings of hexadecimal digits, according to the IEEE 754 standard; IEEE doubles are represented as JSON strings and converted to ordinary doubles internally. Because optimization solvers generally use this format, it provides the most precise way to exchange numerical data with a solver.

The following discussion (Figure 2) highlights some of the key constructs of MOSDEX. Full example is shown in Appendix 1.

*Figure 2: Excerpts of the Transshipment Example in Instance Form*

*Ship Table represents a decision variable artifact*

*Table Schema and Instance data*

*Outputs specified by Function Call fields*

```
25          { "NAME": "ship",
26            "CLASS": "VARIABLE",
27            "KIND": "CONTINUOUS",
28            "SCHEMA": {
29              "FIELDS":
30                ["Name",  "origin",  "destination",  "Column",     "LowerBound", "UpperBound", "Value"],
31              "TYPES":
32                ["STRING","STRING",  "STRING",       "STRING",     "DOUBLE",     "DOUBLE",     "DOUBLE_FUNCTION"]
33            },
34            "INSTANCE": [
35              ["ship", "PITT",    "NE",           "ship_PITT_NE",   0.0,          250.0,   "PrimalValue(Column)"],
36              ["ship", "PITT",    "SE",           "ship_PITT_SE",   0.0,          250.0,   "PrimalValue(Column)"],
37              ["ship", "NE",      "BOS",          "ship_NE_BOS",    0.0,          100.0,   "PrimalValue(Column)" ],
38              ["ship", "NE",      "EWR",          "ship_NE_EWR",    0.0,          100.0,   "PrimalValue(Column)" ],
39              ["ship", "NE",      "BWI",          "ship_NE_BWI",    0.0,          100.0,   "PrimalValue(Column)" ],
40              ["ship", "SE",      "EWR",          "ship_SE_EWR",    0.0,          100.0,   "PrimalValue(Column)" ],
41              ["ship", "SE",      "BWI",          "ship_SE_BWI",    0.0,          100.0,   "PrimalValue(Column)" ],
42              ["ship", "SE",      "ATL",          "ship_SE_ATL",    0.0,          100.0,   "PrimalValue(Column)" ],
43              ["ship", "SE",      "MCO",          "ship_SE_MCO",    0.0,          100.0,   "PrimalValue(Column)" ]
44            ]
45        },
46          { "NAME": "balance",
47            "CLASS": "CONSTRAINT",
48            "KIND": "LINEAR",
49            "SCHEMA": {
50              "FIELDS":
51                ["Name",     "city",   "Row",          "Sense",  "RHS"   ,  "Dual"],
52              "TYPES":
53                ["STRING",   "STRING", "STRING",       "STRING", "DOUBLE", "DOUBLE_FUNCTION"]
54            },
55            "INSTANCE": [
56              ["balance", "PITT", "balance_PITT",    "EQ",     450.0,    "DualValue(Row)"],
57              ["balance", "NE",   "balance_NE",      "EQ",     0.0,      "DualValue(Row)"  ],
58              ["balance", "SE",   "balance_SE",      "EQ",     0.0,      "DualValue(Row)"  ],
59              ["balance", "BOS",  "balance_BOS",     "EQ",     -90.0,    "DualValue(Row)" ],
60              ["balance", "EWR",  "balance_EWR",     "EQ",     -120.0,   "DualValue(Row)" ],
61              ["balance", "BWI",  "balance_BWI",     "EQ",     -120.0,   "DualValue(Row)" ],
62              ["balance", "ATL",  "balance_ATL",     "EQ",     -70.0,    "DualValue(Row)" ],
63              ["balance", "MCO",  "balance_MCO",     "EQ",     -50.0 ,   "DualValue(Row)" ]
64            ]
65          },

75          { "NAME": "balance_shipFrom",
76            "CLASS": "TERM",
77            "KIND": "LINEAR",
78            "SCHEMA": {
79              "FIELDS":
80                ["Row",          "Column",       "Coefficient"],
81              "TYPES":
82                ["STRING",       "STRING",       "DOUBLE"]
83            },
84            "INSTANCE": [
85              ["balance_PITT", "ship_PITT_NE", 1.0          ],
86              ["balance_PITT", "ship_PITT_SE", 1.0          ],
87              ["balance_NE",   "ship_NE_BOS",  1.0          ],
88              ["balance_NE",   "ship_NE_EWR",  1.0          ],
89              ["balance_NE",   "ship_NE_BWI",  1.0          ],
90              ["balance_SE",   "ship_SE_EWR",  1.0          ],
91              ["balance_SE",   "ship_SE_BWI",  1.0          ],
92              ["balance_SE",   "ship_SE_ATL",  1.0          ],
93              ["balance_SE",   "ship_SE_MCO",  1.0          ]
94            ]
95          }
```

**Lines 25-45**: The main elements of a MOSDEX file are its *Tables*. Conceptually, a table is a two-dimensional object with a fixed number of columns, or *fields*, and an indefinite number of rows, or *records*; think of a table in a relational database. *Data* and the modeling artifacts, *Variable*, *Constraint*, *Objective*, and *Term*, are subclasses of Table. A Table's Class and Kind are specified as its first two elements.

The Table **ship** represents the decision variables of the **generalTransshipment** problem. These variables have a two-dimensional *key*, by origin and destination. As in a database, the key uniquely identifies each record in a table.

**Lines 28-33**: As an instance-form Table, **ship** requires an explicit *Schema* which defines the names and types of its fields. The Schema of this Variable includes the key fields and several other fields related to the variable class. In particular, the Column field provides a mapping from the two-dimensional **ship** variable to a column in the tableau of the optimization problem. MOSDEX does not prescribe a particular encoding for the Column field, which may either be a string or an integer; the encoding for this example (a concatenation of the variable name with its keys) has been chosen to make inspection by a human reader easy to decode. The Value field is a placeholder for the solution value computed by the solver; its type is **DOUBLE_FUNCTION**, which indicates a call to a solver method (see discussion in Section 7).

**Lines 34-44**: The Instance array of the **ship** Variable contains actual data. In an *Instance* array, the individual items in each *Record* are unlabeled and can only be parsed using the Table's Schema. The schema's field names, the *Fields* array, are aligned to serve as a visual guide for a human reader, although MOSDEX does not enforce any particular layout of the text. Notice that, in contrast to MPS and other instance formats, MOSDEX does not include coefficient data among the data specifying a Variable. Instead, coefficients are specified in separate Terms Tables, discussed in Lines 75-95 below.

**Lines 30, 32, and 35**: Solution values this example are specified by fields in the modeling artifact tables. For instance, the Value field of the ship table has the type **DOUBLE_FUNCTION** denoting a function call that gets data from the solver. The item **PrimalValue(Column)** in each record will be replaced by its value once the solver has computed an optimal value. As discussed in Section 5 on query-form tables below, MOSDEX also provides a capability to precisely reshape the output data into tables suitable for consumption by down-stream applications.

**Lines 46-65**: The Table **balance** represents the constraints of the **generalTransshipment** problem. These constraints have a one-dimensional key, by city. The Schema and Instance objects of this Constraint are analogous to those discussed for the **ship** Variable. This Table has a one-dimensional key, by city, and a row encoding of the name of the table followed by the key, although again, MOSDEX does not prescribe a particular encoding.

**Lines 75-95**: The Term Table **balance_shipFrom** represents the first of several tables that specify the coefficients of the **generalTransshipment** problem. The name of a Term Table is not prescribed by MOSDEX – any legitimate identifier is acceptable; the choice to use the concatenation of the names of the Constraint and Variable in this example is purely for the convenience of human readers. The schema of a Term Table identifies the Row and Column to which each coefficient

corresponds. Note that Term Tables are also used to specify coefficients for an Objective as well. Because MOSDEX specifies the coefficients in Term Tables separate from the corresponding Variable and Constraint or Objective, it does not favor column-wise or row-wise formulations, unlike other formats, such as MPS or LP.

## 5. Query-Form Tables and the Relational Data Model

The instance-form Tables of MOSDEX shown in the example of the previous Section 4 probably seem familiar to many, especially those who have used MPS or another instance format, since while different in detail, the two standards are similar in structure. However, neither MPS nor instance-form MOSDEX represent data as it exists in its native form. In many cases, especially where an optimization application runs as part of an enterprise decision support system such as MRP, the data originates in an enterprise data store, often a relational database management system. The data streams through a *bridge* from its origin in an enterprise data store into the optimization application. This process is a critical part of designing and operating an optimization application, often entailing substantial software development and computational effort.

Conceptually, mathematical optimization solvers (for linear, mixed integer, and quadratic problems) typically work with a matrix, or *tableau*, representation of a problem internally, and thus it is the most natural way of presenting the data for a particular instance to a solver. However, the tableau form masks a significant aspect of most optimization problems: the tableau (even when implemented as sparse data structures) consists essentially of one- and two-dimensional objects, while data is often multidimensional. Thus, one of the key steps in developing an optimization model is *encoding* the natural multidimensional indices of the data into the one- or two-dimensional indices used by the solver; indeed automatic encoding is one of the main reasons for using an algebraic modeling language. In addition, the data is usually very sparse in practical applications. Solvers, in fact, take advantage of that sparsity in their algorithms to significantly reduce computational effort and speed up solving time. However, data handling ahead of solving can also take advantage of sparsity to reduce the volume of data exchanged with the solver.

Impact of structure, especially sparsity, on data handling up-stream of the solver should not be underestimated by the designers of mathematical optimization models. Sometimes the designer has a lot of control over the format of the source data and can structure it to conform to the requirements of the optimization tableau. However, more often, the source data resides in some kind of enterprise data store that is outside of her control. In that case, the data must be reshaped for input to the solver. Such transformations can be performed by a custom data bridge, usually called the *extraction*, *validation*, *transformation*, and *load* (*EVTL*) process. For realistic problems encountered in practice, the amount of computational effort required to reshape the instance data into tableau format is non-trivial, but it is often unrecognized. It is sometimes said, for example, that modeling languages add a lot of “overhead” in forming an instance for submission to a solver; however, that “overhead” may simply be the unrecognized data transformation effort, which nevertheless must occur, whether in the modeling layer or as part of the EVTL bridge. Another source of misperception arises because “textbook” optimization examples are often so small that the transformations can be done manually, so that the reader is unaware that they have taken place at all. It is thus important for optimization application developers to recognize and account for the data restructuring effort wherever it occurs, rather than simply focusing on solver effort as a benchmark.

As discussed in the paper by Bloom (2017), there is a deep relationship between the structure of the data used in optimization modeling and the artifacts of an optimization model. In fact, as demonstrated in that paper, one can view the modeling layer of an optimization application as transforming the data from its external form in some sort of data store into its internal form in the solver's data structures. These transformations naturally take the form of SELECT queries in SQL. We refer to this relationship between the structure of the data and its relationship to the modeling artifacts of optimization as the *relational data model*.

MOSDEX recognizes the utility of the relational data model by offering an alternative form for specifying a Table, called the *query form*. The following example, Figure 3, shows the same transshipment example of Section 4 but in query form:

*Figure 3: Excerpts of the Transshipment Example in Query Form*

```
27          {
28             "NAME":"ship",
29             "CLASS": "VARIABLE",
30             "KIND": "CONTINUOUS",
31            "QUERY": {
32              "SELECT": [
33                "'ship' AS Name                                          -- STRING",
34                "routes.origin AS origin                                 -- STRING",
35                "routes.destination AS destination                       -- STRING",
36                "CONCAT('ship', '_', origin, '_', destination) AS Column -- STRING",
37                "CAST(0.0 AS DOUBLE) AS LowerBound                       -- DOUBLE",
38                "routes.capacity AS UpperBound                           -- DOUBLE",
39                "'PrimalValue(Column)' AS value                          -- DOUBLE_FUNCTION"
40              ],
41              "FROM": "routes"
42            }
43          },
44          {
45            "NAME": "balance",
46            "CLASS": "CONSTRAINT",
47            "KIND": "LINEAR",
48            "QUERY": {
49              "SELECT": [
50                "'balance' AS Name                                       -- STRING",
51                "cities.city AS city                                     -- STRING",
52                "CONCAT('balance', '_', city) AS Row                     -- STRING",
53                "'EQ' AS Sense                                           -- STRING",
54                "(cities.supply-cities.demand) AS RHS                    -- DOUBLE",
55                "'DualValue(Row)' AS dual                                -- DOUBLE_FUNCTION"
56              ],
57              "FROM": "cities"
58            }
59          },
```

*Ship Table represents a decision variable artifact*

*SQL Query specifies how data are constructed from other Tables*

*MOSDEX schema is specified as part of the Select clause*

```
70          {
71            "NAME": "balance_shipFrom",
72            "CLASS": "TERM",
73            "KIND": "LINEAR",
74            "QUERY": {
75              "SELECT": [
76                "balance.Row AS Row -- STRING",
77                "ship.Column AS Column -- STRING",
78                "CAST(1.0 AS DOUBLE) AS Coefficient -- DOUBLE"
79              ],
80              "FROM": "balance",
81              "JOIN": "ship",
82              "ON": "balance.city = ship.origin"
83              }
84          },
```

*Complex query illustrates joining two tables*

```
175         {
176           "NAME": "shipments",
177           "CLASS": "DATA",
178           "KIND": "OUTPUT",
179           "QUERY": {
180             "SELECT": [
181               "ship.origin AS origin           -- STRING",
182               "ship.destination AS destination -- STRING",
183               "ship.value AS value             -- DOUBLE"
184             ],
185             "FROM": "ship"
186           }
187         },
```

*Shipments Table represents an output data object*
*SQL Query facilitates reshaping results for consumption by downstream applications*

**Lines 27-43:** As in figure 2, the Table **ship** represents the decision variables of the **generalTransshipment** problem. However, in Figure 6, this modeling artifact is defined by an SQL query rather by instance data.

**Lines 31-42:** A MOSDEX Query represents an SQL statement that specifies how the data are constructed from other Tables or from an external database. A Query consists of a list of *clauses*, each of which specifies a *directive*, which is a SQL command, and one or more *predicates*, the arguments of the directive, as the Query for the **ship** Variable illustrates. The **SELECT** clause's predicate specifies how the table's fields are accessed or computed from the fields of the parent table, **routes**. The column encoding is computed as the concatenation of the variable name and the two keys (again, MOSDEX does not specify the column encoding, and the modeler is free to choose any convenient encoding as a string or integer, provided there is no duplication). MOSDEX does not parse SQL, but the directives and predicates must be interpreted directly as valid SQL by the database engine. By tying MOSDEX to SQL, moreover, we take advantage of wide-spread expertise and computational systems available to support relational database systems.

**Lines 32-40:** The **ship** Variable Table does not contain a Schema object, per se. Instead the schema is generated from the query. Specifically, the fields are named in the **AS** part of each item in the **SELECT** clause predicate; furthermore, their types are specified following the "--" symbol, which SQL reads as a comment (as distinguished from comments in JSON, which are not part of its standard).

**Lines 74-83:** The queries in this example illustrate the power of SQL to create compact but efficient data structures for optimization modeling. By joining tables, MOSDEX can use filtering, or slicing, to match data elements with variables and constraints. Such operations are typically computationally intensive and so are best performed using a database engine rather than hand-coded loops that are available in most programming languages.

**Lines 174-186:** The output data table **shipments** is separated from the modeling artifacts. SQL queries facilitate reshaping results for consumption by downstream applications.

Figure 4 shows the MOSDEX results file after a solve using IBM CPLEX. The solver result items are highlighted in green.

*Figure 4: Results after CPLEX Solve*

*Results Module*

*Solver outputs from CPLEX solve*


```
  {
    "NAME" : "results",
    "KIND" : "MODULE",
    "HEADING" : {
      "DESCRIPTION" : [ "Results from General Transshipment Problem" ]
    },
    "TABLES" : [
      {
      "NAME" : "shipments",
      "CLASS" : "DATA",
      "KIND" : "OUTPUT",
      "SCHEMA" : {
        "FIELDS" : [ "origin", "destination", "value" ],
        "TYPES" :  ["STRING", "STRING",      "DOUBLE" ] },
      "INSTANCE" : [
                   [ "PITT",    "NE",          250.0 ],
                   [ "PITT",    "SE",          200.0 ],
                   [ "NE",      "BOS",          90.0 ],
                   [ "NE",      "EWR",         100.0 ],
                   [ "NE",      "BWI",          60.0 ],
                   [ "SE",      "EWR",          20.0 ],
                   [ "SE",      "BWI",          60.0 ],
                   [ "SE",      "ATL",          70.0 ],
                   [ "SE",      "MCO",          50.0 ] ]
      },
      {
      "NAME" : "objective",
      "CLASS" : "DATA",
      "KIND" : "OUTPUT",
      "SCHEMA" : {
        "FIELDS" : [  "cost" ],
        "TYPES" :  [  "DOUBLE" ] },
      "INSTANCE" : [ [ 1819.0 ] ]
      } ]
  } ]
}
```

## 6. Indexing, Scaling and Structure in MOSDEX

Optimization models frequently (in fact, almost always) use indexing to represent groups of related variables or constraints, a common practice in mathematics. For example, instead of giving each variable a distinct name, such as *x, y, z*, one uses an index like this $x_1, x_2, x_3$. To a solver, a problem instance has a representation (conceptually) as a two-dimensional array, the *tableau*, in which the column dimension represents the variables and row dimension represents the constraints and objectives (in fact, of course, the tableau is sparse, so that it is not literally represented as a matrix in memory). However, in many models, the data behind these artifacts has more dimensions. For instance, in the transshipment model discussed above (Equation 1), the ship variables themselves have two dimensions, the origin and the destination. Thus, one of the key steps in developing an optimization model is encoding the natural multidimensional indices of the data into the one- or two-dimensional indices used by the solver.

Different solver implementations use various means to identify columns and rows. Some allow symbolic names while others use integers. Instance formats often deal with indexing in the way they assign identifiers to the columns and rows. The encoding is often ad hoc, not standardized, and many times undocumented. Furthermore, encoded identifiers are cumbersome to work with; for instance, an operation such as *slicing*, say collecting all the shipments with a common destination, are hard to do with symbolic identifiers and impossible with integer encoding.

Algebraic modeling languages point the way to a more general view of indexing. These languages introduce the idea that an index is an element of a set. An index set need not consist solely of positive integers (as is traditional in mathematics), but instead may comprise other scalar values, such as strings, or tuples of values (e.g. pairs, triples, etc.). Furthermore, index sets can be constructed from other sets through operations such as joins, unions, or slicing. A slice is a subset of tuples in which certain components have fixed values. The algebraic languages then map these abstract index sets to actual row and column identifiers in a process which is transparent to the modeler.

In our view, the representation and construction of abstract index sets is an aspect of modeling that is too often neglected in optimization education. This lacuna is reinforced by the traditional mathematical notation used specifying optimization models, where indices are relegated to subscripts (and sometimes superscripts) and where the index sets are either encoded as integers or are described (somewhat vaguely) in text rather than in specific mathematical formulas. Yet, perhaps surprisingly to many readers, there is already a rich mathematical theory that supports abstract indexing, namely *relational algebra*, and its widely used implementation in Structured Query Language (SQL); see, for example, the article on Relational Algebra (Wikipedia, 2022).

MOSDEX uses an indexing strategy based on SQL. MOSDEX does not dictate a particular naming pattern for column and row identifiers. The **generalTransshipment** example uses a concatenation of the name of the Table with its keys, but this method is a convenience rather than a requirement of MOSDEX. Rather, a MOSDEX modeling artifact table has a record schema that includes both the primary keys of the underlying data (e.g. the origin and destination of the **ship** variable) and the column or row identifier as a foreign key (see Figure 3, line 36). Using SQL, MOSDEX tables then can implement index set operations using the primary keys and can access the solver-specific modeling artifacts through the foreign key.

Use of SQL enables MOSDEX data files to scale more compactly than instance formats, which scale linearly in the number of variables and constraints. Consider the ship variable in the transshipment model discussed above. In instance form (Figure 2, lines 25-45), it has one record for each route; however, in query form (Figure 3, lines 27-43), it has a single SQL statement that does not change size as the number of routes changes. While it may seem that we have simply moved the large instance array upstream from the modeling artifact to the input data, in fact the input data may not even reside in MOSDEX but rather in a table of the enterprise data store accessed by the optimization application.

Furthermore, the difference in scaling also affects the coefficients in the **balance_shipFrom** term. In instance form (Figure 2, lines 75-94), it has the constant coefficient 1.0 for each route out of each origin, and it scales linearly in the number of such combinations. However, in query form (Figure 3, lines 70-84), again it has a single SQL statement that does not change size as the number of origins and routes changes, and there is no upstream data table since the coefficient entries are constants. One could argue that such a flow conservation constraint in a network model could be handled in an instance format by a special structure (and indeed, many offer a special "network" syntax), especially since it is very common, but why should we proliferate special structures when a general artifact will do?

Use of SQL also enables MOSDEX to represent the structure of an optimization problem. As is well-known, most non-trivial mathematical optimization models have a great deal structure. Beyond the obvious fact of data sparsity, many models have repeating blocks of variables and constraints that differ only in their location in the row and/or column spaces of the problem tableau. Solvers almost always take advantage of sparsity, but specialized solution algorithms, such as decomposition, can be designed to exploit other structural constructs, which can greatly speed-up solving and, in some cases, might be the only feasible way to find a solution. Stochastic programs, in particular, represent the one of the most important categories of large problem instances, for which development of specialized algorithms has been a major area for research.

As is evident from the two examples discussed above (Figure 2 and Figure 3), MOSDEX tables mirror the block structure of the underlying model. MOSDEX modules are intended to support decomposition and specialized solver algorithms (see the discussion in section 8).

Furthermore, not only can explicit structure enable specialized solution algorithms, but it can also facilitate analysis of a model for purposes of validation, debugging, and qualitative explanation of solution properties to decision makers. The importance of these activities and the kind of software tools that support them are the subject of the paper by Saltzman (2021), who surveys the prior contributions of Greenburg. While Greenburg's work was mostly based on MPS, we believe that analogous tools using the more powerful artifacts of MOSDEX would enhance their utility.

The use of SQL to create an optimization problem is more than a practical matter of software engineering; the underlying relational algebra provides a computational model for data flows surrounding an optimization solver, clearly elucidating the complexity of the transformations involved, which may be useful in other, more theoretical contexts.

## 7. Retrieving Solutions and Formatting Output in MOSDEX

Optimization solvers can produce a plethora of information about the solution of a model, such as

- optimal primal value, reduced cost, and basis status of a variable,

- optimal dual value and slack of a constraint, and
- optimal value of the objective function.

MOSDEX takes a simple approach to specifying solution results that does not require a lot of special syntax.

In MOSDEX, solution items are denoted as part of the schema of a modeling artifact table, with one of the special function types denoted as

- **DOUBLE_FUNCTION**,
- **IEEEDOUBLE_FUNCTION**,
- **INTEGER_FUNCTION**,
- **STRING_FUNCTION**,

where the prefix indicates the type of the item returned. Their field names are not pre-specified in the record schema in order to leave the opportunity for the MOSDEX user to choose her own. The value assigned to each such field, either in an instance or in a query, is a *function call* string of the form

**"functionName(argumentField1, ...argumentFieldN)"**

where the **argumentFields** are the names of the fields in the current record where the arguments are found (usually, there is only one argument field, a column or a row, in a function call); see Figure 3, line 39.

The **functionName** links to a method of the solver's API that retrieves the relevant value from its solution. After the solve, MOSDEX creates a solution output table from the modeling artifact, in which that value is substituted for the function call string, and it replaces the function type with the corresponding return type in the output table's schema.

In order to feed to downstream stages of the application stack, it might be desirable to transform the solution (and other information from the upstream stages) by creating other output tables from queries of the solution tables. Thus, MOSDEX allows the user to specify the format of the output tables using SQL.

## 8. Additional Capabilities of MOSDEX

Modern optimization solvers support a variety of problem types beyond purely linear programs. As a result, a data exchange format needs to support these additional capabilities, with general artifacts that can support a variety of solver-specific syntax that are not readily extensible. MOSDEX is designed both to accommodate many additional capabilities and also be extensible in a systematic fashion. Among the features of MOSDEX that provide these extensions are as follows:

### 8.1 Linear, Mixed Integer, and Quadratic Problems

The Kind element of a Variable can be designated as Continuous, Integer, or Binary. The Kind element of a Term can be designated as Linear or Quadratic. A standard schema is defined for each type of object. A MOSDEX parser should create the corresponding objects of the solver's API. Here are some examples using these capabilities:

- Simple linear program – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/volsay_2-0.json [ii]

- Network (instance form) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/net1b_2-1.json [iii]
- Network (query form) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/net1a_2-1.json [iv]
- LP with lagged variables – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/sailco_2-0.json [v]
- Warehouse location (MIP) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/warehousing_2-0.json [vi]
- Traffic Network (QP) – https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/trafficNetworkQP_2-0.json[vii]

Although not implemented yet, MOSDEX is extensible to special structures such as special ordered sets, indicator variables, and so on.

**8.2 Modular Structures**

Modular structures arise when two or more Modules interact with each other. Perhaps the most familiar modular structure is *decomposition*, in which one problem, designated the *master*, interchanges information iteratively with one or more *subproblems*. However, modular structures can also occur in stochastic optimization and in combined optimization/simulation applications. As an example consider a stochastic version of the warehouse location problem[viii]. This model is formulated as a two-stage stochastic program with recourse. The first stage determines the locations and capacities of warehouses to open. The second stage determines how much product to ship to each store, where the demand at each store is uncertain. Solved by decomposition (often Benders decomposition), the master problem comprises the first stage, and the subproblems, one for each demand scenario, comprise the second stage. While this aspect is still under development, our current thinking proposes several constructs that enable building modular structures in MOSDEX. First, each modeling Module (that is, any containing modeling artifacts) is associated with its own solver instance, for example, one processor node in a distributed solver. Second, a Table can be initialized or updated from another Module, in order to facilitate communication among them.

Figure 5 below illustrates additional MOSDEX constructs for modular structures relevant for the subproblems of the stochastic warehouse location problem.

*Figure 5: Extract of Subproblem Module from Stochastic Warehouse Location Model*

```
"FOR_EACH": {
  "scenario": ["scenarios"]
},
"TABLES": [
  {
    "NAME": "scenarios",
    "CLASS": "STRUCTURE",
    "QUERY": {
      "SELECT": [
        "'subproblem' AS module               -- STRING",
        "2 AS stage                           -- INTEGER",
        "demands.id AS id                     -- INTEGER",
        "demands.weeks/52.0 AS probability    -- DOUBLE",
        "'masterProblem' AS parent            -- STRING",
        "'solveStatus(module, id)' AS status  -- STRING_FUNCTION"
      ],
      "FROM": ["demands"]
    }
  },
  {
    "NAME": "ship",
    "CLASS": "VARIABLE",
    "KIND": "CONTINUOUS",
    "QUERY": {
      "SELECT": [
        "'ship' AS Name                           -- STRING",
        "scenario.id AS id                        -- INTEGER",
        "routes.location AS location              -- STRING",
        "routes.store AS store                    -- STRING",
        "ID4('ship', id, location, store) AS Column -- STRING",
        "0.0D AS LowerBound                       -- DOUBLE",
        "1.0D AS UpperBound                       -- DOUBLE",
        "PrimalValue('Column') AS Value           -- DOUBLE_FUNCTION"
      ],
      "FROM": [ "scenario", "routes" ]
    }
  },
...
]
```

*FOR_EACH replicates a Module*

*STRUCTURE Table indicates relationships among Modules*

*Placeholder scenario refers to a record in the Structure Table*

As a third new construct, MOSDEX provides a **STRUCTURE** class, a subclass of Table, that contains information relating to its relationship to other Modules. MOSDEX offers some flexibility regarding the schema of a Structure Table; however, in many cases, the solver algorithm will dictate its schema and content. For instance, in this example, the Structure Table will has a field called **module** to distinguish the master problem from the subproblems, a field called **stage** to distinguish the stage or time, and a field called **id** to distinguish among the subproblems by scenario or state. Many modular models organize their various subproblems in a tree structure, so the field **parent** identifies its parent Module.

As a fourth new construct, MOSDEX provides a capability to replicate Modules. One of the key features of modular structure is the replication of a block of variables and constraints at different locations in the tableau. The stochastic warehouse location example has multiple versions of a minimum cost flow problem, one for each realization of the stores' demands. MOSDEX can represent this replication by parameterizing a Module so that one Module can provide a template for multiple blocks. The parameterization is represented by an object in the Module denoted by the key word **FOR_EACH**.

In the **FOR_EACH** object in Figure 5, **scenario** represents a placeholder for a row in the structure table **scenarios**. MOSDEX then interprets the Module containing the **FOR_EACH** as a family of replicas, one

for each of the records cited in the **FOR_EACH** object, and the Tables in the Module can use the record in their queries to distinguish among the replicas.

The placeholder **scenario** is used in the queries creating the various modeling artifacts that are specific to the subproblem. For instance, there is one instance of the **ship** variable for each demand scenario, so the scenario id is included as a field in the variable's Select query.

It is important to understand that MOSDEX itself does not provide an algorithm for solving modular problems; that is the province of a solver. For example, setting up a column generation structure in MOSDEX will not be sufficient for an ordinary solver of linear programs to execute the decomposition. The solver part of the software stack must have a control structure that alternately solves the master problem and subproblems, checks the convergence criterion, and terminates the solve process. Furthermore, the solver algorithm's requirements will determine, in part, how the problems are set up in MOSDEX. What MOSDEX does provide are standards to specify the Modules that the solver executes, how they exchange data and coordinate during execution.

### 8.3 Nonlinear Problems

This experimental aspect of MOSDEX is an attempt to support nonlinear models. It would require a suitable optimization solver and a software bridge capable to construct the nonlinear functions specified by MOSDEX. As an example, consider the Traffic Network[vii] problem, a quadratic program. Figure 6 shows the two nonlinear terms as quadratics specified in the current version of MOSDEX. Note that they both involve references (i.e. foreign keys) to two columns, one for each decision variable factor.

*Figure 6: Quadratic Terms Example*

```
{
  "NAME": "delayDef_flowSq",
  "CLASS": "TERM",
  "KIND": "QUADRATIC",
  "QUERY": {
    "SELECT": [
      "delayDef.Row AS Row              -- STRING",
      "flow.Column AS Column            -- STRING",
      "flow.Column AS Column2           -- STRING",
      "roads.sensitivity AS Coefficient -- DOUBLE"
    ],
    "FROM": [ "delayDef" ],
    "JOIN": [ "flow" ],     "USING": [ "(origin, destination)" ],
    "JOIN#2": [ "roads" ],  "USING#2": [ "(origin, destination)" ]
  }
},
{
  "NAME": "delayDef_delay_slack",
  "CLASS": "TERM",
  "KIND": "QUADRATIC",
  "QUERY": {
    "SELECT": [
      "delayDef.Row AS Row                  -- STRING",
      "delay.Column AS Column               -- STRING",
      "slack.Column AS Column2              -- STRING",
      "CAST(-1.0 AS DOUBLE) AS Coefficient  -- DOUBLE"
    ],
    "FROM": [ "delayDef" ],
    "JOIN": [ "delay" ],    "USING": [ "(origin, destination)" ],
    "JOIN#2": [ "slack" ],  "USING#2": [ "(origin, destination)" ]
  }
}
```

While an extension of MOSDEX for general nonlinear terms is still under development, our current thinking proposes two new constructs to enable building nonlinear models. First, we define, in addition to instance- and query-form tables, a *formula* form, with syntax illustrated in Figure 7.

*Figure 7: Nonlinear Terms Example*

*FORMULA defines a nonlinear calculation*

```
{
  "NAME": "delayDef_flowSq",
  "CLASS": "TERM",
  "KIND": "NONLINEAR",
  "FORMULA": {
    "CALCULATE": {"x": "#sensitivity * #flow^2"},
    "VARIABLES": ["flow"],
    "PARAMETERS": ["sensitivity"]
  },
  "QUERY": {
    "SELECT": [
      "delayDef.Row AS Row              -- STRING",
      "flow.Value AS flow               -- DOUBLE",
      "roads.sensitivity AS sensitivity -- DOUBLE",
      "'Calculate(x)'                   -- DOUBLE_FUNCTION"
    ],
    "FROM": [ "delayDef" ],
    "JOIN": [ "flow" ],     "USING": [ "(origin, destination)" ],
    "JOIN#2": [ "roads" ],  "USING#2": [ "(origin, destination)" ]
  }
},
{
  "NAME": "delayDef_delay_slack",
  "CLASS": "TERM",
  "KIND": "NONLINEAR",
  "FORMULA": {
    "CALCULATE": {"y": "#Coefficient * #slack * #delay"},
    "VARIABLES": ["slack", "delay"],
    "PARAMETERS": ["Coefficient"]
  },
  "QUERY": {
    "SELECT": [
      "delayDef.Row AS Row                  -- STRING",
      "delay.Column AS delay                -- STRING",
      "slack.Column AS slack                -- STRING",
      "CAST(-1.0 AS DOUBLE) AS Coefficient  -- DOUBLE",
      "'Calculate(y)'                       -- DOUBLE_FUNCTION"
    ],
    "FROM": [ "delayDef" ],
    "JOIN": [ "delay" ], "USING": [ "(origin, destination)" ],
    "JOIN#2": [ "slack" ], "USING#2": [ "(origin, destination)" ]
  }
}
```

Second, we propose a specific syntax for the **FORMULA** object in MOSDEX, which includes several fields. The **CALCULATE** field defines one or more symbolic expressions, which might be as simple as a single unary or binary operation or much more complex combinations of such operations. As with SQL queries, we don't intend for MOSDEX to have a parser for such calculations; instead, as with SQL, we expect to rely on a widely utilized library, such as Spring Expression Language (SpEL).

The **VARIABLES** field identifies the linkages with the decision variables of the optimization problem; they would be the Column identifiers in a Variables table. The **PARAMETERS** field identifies other data items that do not depend on the optimization result. The **CALCULATE** field identifies the results of the calculation. All these fields' components would also appear in an instance or query object in the same

table, thus linking the formula to the other modeling artifacts. The double function **`Calculate`** links the formula result to the query.

Formula-form Tables behave like other Tables (i.e. they each have a defined schema, and they can have Instance or Query form). The main innovation in MOSDEX is that a formula-form Table has dimensionality as specified by its key fields. Thus MOSDEX implements families of formulas that correspond to the structure of the optimization problem that it represents. This represents yet another aspect of the scaling in MOSDEX discussed in Section 6. Other formats for nonlinear expressions, notably *nl* format (Gay, 2005), resemble instance format, in which each instance of a nonlinear term requires its own specification of the formula, even if the underlying calculation is the same.

One issue that arises, almost uniquely, in optimization models is the need to evaluate derivatives of the nonlinear expressions. The current state of the art on this issue uses the *syntax tree* of the expression to symbolically compute its derivatives (see (Bell, 2022)). Therefore, an expression library should have an accessible syntax tree; SpEL meets this requirement, although it does not appear to have a symbolic differentiation capability.

Again, MOSDEX itself does not provide an algorithm for solving nonlinear problems; that is the province of a solver. The solver algorithm's requirements will determine, in part, how the problems are set up in MOSDEX. What MOSDEX does provide are standards to specify the expressions that the solver evaluates.

## 9. Architectural Considerations

MOSDEX files serve as repositories for the data (and modeling artifacts, where needed) that support optimization-based decision applications. They are, in effect, the source and destination ends of a process, or *bridge*, that transforms information into decisions. In this section, we consider that process and how MOSDEX supports it.

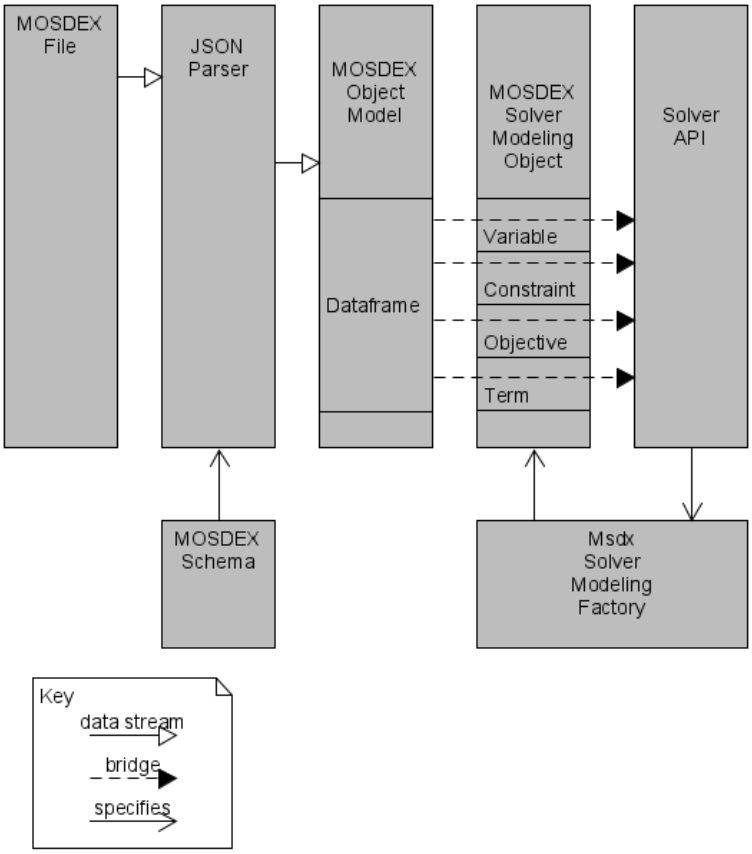
The main steps of the data flow for an optimization-based application are as follows (see Figure 1: Data Flow in an Optimization Application):

1. Data are extracted from one or more sources and are validated to insure that they represent a well-formed problem instance. Validity checks could include, for example, coefficients falling within acceptable limits or incidence maps ensuring topological connectivity of a network.
2. Data are transformed and reshaped to conform with the syntax of the optimization model. For example, cost data must be associated with activities whose levels will be determined by variables to be optimized and resource data must be associated with constraint limits.
3. Data populate the internal data structures of the optimization solver. How this step occurs is determined by the application programming interface (API) of the solver, which presents its public face. Different solvers have their own unique APIs, which can range from simple matrix arguments in calls to the solver routines to sophisticated object-oriented class libraries.
4. The solver executes its optimization algorithm, and the results are exposed through its API.
5. Data retrieved from the solver are transformed and reshaped to conform with the requirements of the applications that consume it, which can include both automated processes and interfaces for human users.

In addition to documenting the MOSDEX syntax for data exchange, we have created a *reference architecture* (*RefArc*) for the transformation of MOSDEX data to and from a solver, as illustrated in Figure 8.

*Figure 8: MOSDEX Reference Architecture*




The MOSDEX RefArc has been implemented in Java using Apache Spark for the Dataframes, which hold the instance data and execute the queries, and IBM CPLEX as the solver [ix]

In the first two steps the data flow (Figure 1: Data Flow in an Optimization Application), MOSDEX serves as the format that enables standardization. MOSDEX's use of query-form tables provides a means to document the execution of extraction, validation, and transformation of data as it moves through this process. It does so in a platform independent manner.

Typically, the results of the second step are captured in internal objects of some kind of business application platform built on top of a programming language like Java or Python. Referring to Figure 8, a MOSDEX *parser* translates the MOSDEX files into these internal objects, which mirror the MOSDEX structure. That is, for example, a MOSDEX Table has a counterpart Table class in the underlying programming language, say Java. This set of classes collectively are called the MOSDEX *Object Model*.

The third step transforms the MOSDEX Object Model into classes of the solver's API (Figure 8). In this step, MOSDEX serves to standardize the object model.

The fourth and fifth steps reverse the three input steps by transforming the API classes of the solver back into the MOSDEX Object Model and then providing access to them by the consuming applications. As in the first two steps, the MOSDEX standard serves to document the execution of these transformations.

As noted above, the data transformations, or *bridges*, upstream and downstream of the solver are both essential to an optimization-based application and also potentially computationally burdensome. Thus, economy and efficiency of these bridges are often critical considerations in the application design, but ones that are often neglected by domain experts in optimization. We believe that standardizing data exchange along these bridges will lead to advances in both algorithm design and application design.

One such advance that is rapidly developing is the adaptation to optimization of parallelization technologies in information processing. The latest wave of these technologies had its origin in the need to manage massive data sets for internet search and machine learning, which resulted in development of the Hadoop and Spark software libraries, among others. They provide the capability to distribute processing across clusters of computers, operating in parallel.

Spark (Apache Spark), in particular, uses an abstraction for a bridge span called a *resilient distributed dataset* (RDD), which partitions data items and distributes them onto the nodes of a cluster and which enables certain operations on a span, called *transformations*, to occur in parallel. The prototype transformation is *map*, which applies a function to each item of data without aggregating the results. Transformations are *lazy*, in the sense that they are not executed immediately but instead are queued until a triggering operation, or *terminal action*, requires returning a result to the calling program, or *driver*. The queue can then be reordered to optimize computing all of its transformations. The prototype terminal action is *reduce*, which aggregates the data items according to some function, such as a sum, and returns a final result to the driver. A terminal action is often the final span of the bridge.

Spark can handle structured data (that is, data with a schema) using a *dataframe*, which is built on top of RDD and inherits the ability to parallelize certain transformations. Dataframes can process their items using SQL queries, executed in parallel, which makes them useful to hold MOSDEX objects containing very large Instances. Very large optimization problem instances arise in a variety of domains, such as stochastic programming for example. In these domains, there is often an intimate relationship between the structure of the data and the optimization algorithm applied to it. Thus, MOSDEX is particularly suited to specifying both the data and the modeling artifacts in a standard manner, facilitating algorithm development.

Using parallelization makes it possible to build the upstream and downstream data bridges surrounding the optimization solver as *streams*. In a stream, data items are processed sequentially as they are generated and do not reside in memory until the entire bridge has been completed. Ideally, the data rests only at the source and destination of the bridge and not at intermediate spans. Thus, streaming is potentially both efficient and economical when dealing with very large data sets. We intend for MOSDEX to support and facilitate streaming architectures for optimization-based applications.

## 10. Conclusion

This paper has proposed a new standard called MOSDEX that improves on existing instance formats, such as MPS, for data exchange with optimization solvers. MOSDEX is based on several principles:

independence from and support for multiple optimization solvers and their APIs and for multiple algebraic modeling languages, model/data separation, relational data modeling, and incorporation of standard optimization modeling artifacts. MOSDEX uses the widely adopted JSON data format standard to take advantage of JSON support in a variety of programming languages including Java, C++, Python, and Julia. The paper has demonstrated the principles of MOSDEX through examples taken from well-known optimization problems.

**References**


AMPL Optimization Inc. (n.d.). *AMPL*. Retrieved from https://ampl.com/

*Apache Spark*. (n.d.). Retrieved from https://spark.apache.org/

Atamturk, A., Johnson, E. L., Linderoth, J. T., & Savelsbergh, M. (2000). A Relational Modeling System For Linear and Integer Programming. *Operations Research, 48 (6)*, 846–857.

Bell, B. (2022). *A C++ Algorithmic Differentiation Package*. Retrieved from CppAD: https://coin-or.github.io/CppAD/doc/cppad.htm

Bloom, J. A. (2017). *Optimization Modeling and Relational Data.* Retrieved from https://github.com/JeremyBloom/Optimization---Sample-Notebooks/blob/master/Optimization%2BModeling%2Band%2BRelational%2BData%2Bpub.ipynb

Colombo, M., Grothey, A., Hogg, j., & et al. (2009). A structure-conveying modelling language for mathematical and stochastic programming. *Math. Prog. Comp., 1*, 223–247. Retrieved from https://doi.org/10.1007/s12532-009-0008-2

Fourer, R. (1997). Database structures for mathematical programming models. *Decision Support Systems, 20*, 317-344.

Fourer, R., Gay, D. M., & Kernigan, B. W. (2003). *AMPL: A Modeling Language for Mathematical Prgramming.* Pacific Grove, CA: Thomson. Retrieved from https://ampl.com/BOOK/EXAMPLES/EXAMPLES2/net1.mod

Fourer, R., Ma, J., & Martin, K. (2010). Optimization Services: A Framework for Distributed Optimization. *Operations Research, 58*(6), 1624-1636. Retrieved from https://doi.org/10.1287/opre.1100.0880

Fourer, R., Ma, J., and Martin, K. . (2010). OSiL: An Instance Language for Optimization. *Comput. Optim. Appl., 45*(1), 181–203. Retrieved from Fourer, R., Ma, J. & Martin, K. OSiL: An instance language for optihttps://doi.org/10.1007/s10589-008-9169-6

GAMS Development Corp. (n.d.). *GAMS*. Retrieved from https://www.gams.com/

Gassmann, H., Ma, J., & Martin, K. (2011). Instance Formats for Mathematical Optimization Models. Retrieved from https://doi.org/10.1002/9780470400531.eorms0411

Gay, D. M. (2005). *Writing .nl Files.* Sandia National Laboratories. Retrieved from https://ampl.github.io/nlwrite.pdf

IBM Corporation. (n.d.). *OPL*. Retrieved from https://www.ibm.com/docs/en/icos/12.8.0.0?topic=opl-optimization-programming-language

*JSON*. (n.d.). Retrieved from http://json.org/

*JSON Schema*. (n.d.). Retrieved from http://json-schema.org/

Saltzman, M. J. (2021). Software for an Intelligent Mathematical Programming System. In A. (. Holder, & H. J. Greenberg, *International Series in Operations Research & Management Science* (pp. 47-63). Springer. Retrieved from https://ideas.repec.org/h/spr/isochp/978-3-030-56429-2_3.html

*SpEL*. (n.d.). Retrieved from Spring Expression Language: https://docs.spring.io/spring-framework/docs/4.3.10.RELEASE/spring-framework-reference/html/expressions.html

Wikipedia. (2018). *MPS format.* Retrieved from https://en.wikipedia.org/wiki/MPS_(format)

Wikipedia. (2022). *Relational Algebra.* Retrieved from Relational Algebra: https://en.wikipedia.org/wiki/Relational_algebra

**About the Authors**

Dr. Jeremy A. Bloom retired in 2017 after a 40-year career in operations research. Most recently, he was responsible for IBM's Decision Optimization Center product, an application development and deployment platform using IBM's CPLEX optimization solver and its OPL algebraic modeling language. Prior to joining IBM, he worked in technical sales and product marketing at ILOG before its acquisition by IBM. Prior to joining ILOG, Dr. Bloom managed programs at the Electric Power Research Institute in power delivery asset management, retail market analysis and resource management for the restructured power industry, distributed energy resources, and integrated resource planning. While at EPRI, he was part of the leadership team of a spin-out providing information and market research for retail energy markets, and he was responsible for technical leadership of a proposal to manage California's energy efficiency market transformation programs. Earlier, he spent a significant part of his career at General Public Utilities, where he was responsible for resource planning and demand-side management, including leading the company's first efforts to procure demand-side resources through competitive bidding. He began his career teaching operations research at Cornell University. Dr. Bloom received his undergraduate degree in electrical engineering at Carnegie-Mellon University and his graduate degrees in operations research from the Massachusetts Institute of Technology.

Other co-authors

**Appendix 1. Instance-Form Example**

This appendix gives the full MOSDEX File for the **generalTransshipment** model in instance form (Figure 9 and Figure 10).

A MOSDEX *File* consists of one of more *Modules*. (We use the term file generically for any input source.) The example has just one module named **generalTransshipment**. It is useful to think of a Module as a self-contained presentation of the data and modeling artifacts for a mathematical optimization problem; however, MOSDEX can actually accommodate more general structures. For instance, when model/data separation is used, the data may be presented in one or more separate Modules without modeling artifacts.

**Lines 6-23**: A *Heading* element provides documentation for human readers. A Heading is required for a Module, although only the Description is mandatory. In this example, the Heading also contains a Math expression of the problem, in any suitable language, such as OPL, AMPL, or GAMS. The information in the Heading, including the Math object, are not otherwise processed by MOSDEX.

**Lines 24-138**: The main elements of the Module are its *Tables*. Conceptually, a table is a two-dimensional object with a fixed number of columns, or *fields*, and an indefinite number of rows, or *records*; think of a table in a relational database. *Data* and the modeling artifacts, *Variable*, *Constraint*, *Objective*, and *Term*, are subclasses of Table. A Table's Class and Kind are specified as its first two elements.

**Lines 25-45**: The Table **ship** represents the decision variables of the **generalTransshipment** problem. These variables have a two-dimensional *key*, by origin and destination. As in a database, the key uniquely identifies each record in a table.

**Lines 28-33**: As an instance-form Table, **ship** requires an explicit *Schema* which defines the names and types of its fields. The Schema of this Variable includes the key fields and several other fields related to the variable class. In particular, the Column field provides a mapping from the two-dimensional **ship** variable to a column in the tableau of the optimization problem. MOSDEX does not prescribe a particular encoding for the Column field, which may either be a string or an integer; the encoding for this example (a concatenation of the variable name with its keys) has been chosen to make inspection by a human reader easy to decode. The Value field is a placeholder for the solution value computed by the solver; its type is DOUBLE_FUNCTION, which indicates a call to a solver method (see discussion in section 7).

**Lines 34-44**: The Instance array of the **ship** Variable contains actual data. In an *Instance* array, the individual items in each *Record* are unlabeled and can only be parsed using the Table's Schema. The schema's field names, the *Fields* array, are aligned to serve as a visual guide for a human reader, although MOSDEX does not enforce any particular layout of the text. Notice that, in contrast to MPS and other instance formats, MOSDEX does not include coefficient data among the data specifying a Variable. Instead, coefficients are specified in separate Term Tables, discussed in lines 75-95 below.

**Lines 46-65**: The Table **balance** represents the constraints of the **generalTransshipment** problem. These constraints have a one-dimensional key, by city. The Schema and Instance objects of this Constraint are analogous to those discussed for the **ship** Variable. This Table has a one-dimensional key, by city, and a

row encoding of the name of the table followed by the key, although again, MOSDEX does not prescribe a particular encoding.

**Lines 66-74**: The Table **totalCost** represents the objective function of the **generalTransshipment** problem. This table has a single row. Note that the Objective is assigned a Row identifier.

**Lines 75-95**: The Term Table **balance_shipFrom** represents the first of several tables that specify the coefficients of the **generalTransshipment** problem. The name of a Term Table is not prescribed by MOSDEX – any legitimate identifier is acceptable; the choice to use the concatenation of the names of the Constraint and Variable Tables in this example is purely for the convenience of human readers. The schema of a Term Table identifies the Row and Column to which each coefficient applies. Note that Term Tables are also used to specify coefficients for an Objective as well. Because MOSDEX specifies the coefficients in separate Tables from the corresponding Variable and Constraint or Objective, it does not favor column-wise or row-wise formulations, unlike other formats, such as MPS or LP.

**Lines 30, 32, and 35**: Solution values this example are specified by Fields in the modeling artifact tables. For instance, the Value field of the ship table has the type **DOUBLE_FUNCTION** denoting a function call that gets data from the solver. The item **PrimalValue(Column)** in each record will be replaced by its value once the solver has computed an optimal value. As discussed in the section on query-form tables below, MOSDEX also provides a capability to precisely reshape the output data into tables suitable for consumption by down-stream applications.

*Figure 9: Full Transshipment Example in Instance Form*

```
{
  "SYNTAX": "MOSDEX/MOSDEX v2/MOSDEXSchemaV2-0.json",
  "MODULES": [
    { "NAME": "generalTransshipment",
      "CLASS": "MODEL",
      "HEADING": {
        "DESCRIPTION": [
          "General Transshipment Problem",
          "instance form", "with a function calls for output",
          "MOSDEX 2-0 Syntax"
        ],
        "VERSION": "net1b 2-1",
        "REFERENCE": ["https://ampl.com/BOOK/EXAMPLES/EXAMPLES2/net1.mod"],
        "AUTHOR": ["Jeremy A. Bloom (jeremyblmca@gmail.com)"],
        "NOTICES": ["Copyright 2019 Jeremy A. Bloom"],
        "MATH": [
          "var Ship {(i,j) in ROUTES} >= 0, <= capacity[i,j]; # packages to be shipped",
          "minimize Total_Cost: sum {(i,j) in ROUTES} cost[i,j] * Ship[i,j];",
          "subject to",
            "Balance {k in CITIES}: ",
              "sum {(k,j) in ROUTES} Ship[k,j] - sum {(i,k) in ROUTES} Ship[i,k] = supply[k] - demand[k];"
        ]
      },
      "TABLES": [
        { "NAME": "ship",
          "CLASS": "VARIABLE",
          "KIND": "CONTINUOUS",
          "SCHEMA": {
            "FIELDS":
              ["Name",  "origin",  "destination",  "Column",     "LowerBound", "UpperBound", "Value"],
            "TYPES":
              ["STRING","STRING",  "STRING",       "STRING",     "DOUBLE",     "DOUBLE",     "DOUBLE_FUNCTION"]
          },
          "INSTANCE": [
            ["ship", "PITT",    "NE",          "ship_PITT_NE",   0.0,          250.0,   "PrimalValue(Column)"],
            ["ship", "PITT",    "SE",          "ship_PITT_SE",   0.0,          250.0,   "PrimalValue(Column)"],
            ["ship", "NE",      "BOS",         "ship_NE_BOS",    0.0,          100.0,   "PrimalValue(Column)" ],
            ["ship", "NE",      "EWR",         "ship_NE_EWR",    0.0,          100.0,   "PrimalValue(Column)" ],
            ["ship", "NE",      "BWI",         "ship_NE_BWI",    0.0,          100.0,   "PrimalValue(Column)" ],
            ["ship", "SE",      "EWR",         "ship_SE_EWR",    0.0,          100.0,   "PrimalValue(Column)" ],
            ["ship", "SE",      "BWI",         "ship_SE_BWI",    0.0,          100.0,   "PrimalValue(Column)" ],
            ["ship", "SE",      "ATL",         "ship_SE_ATL",    0.0,          100.0,   "PrimalValue(Column)" ],
            ["ship", "SE",      "MCO",         "ship_SE_MCO",    0.0,          100.0,   "PrimalValue(Column)" ]
          ]
      },
        { "NAME": "balance",
          "CLASS": "CONSTRAINT",
          "KIND": "LINEAR",
          "SCHEMA": {
            "FIELDS":
              ["Name",     "city",   "Row",          "Sense",  "RHS"   ,  "Dual"],
            "TYPES":
              ["STRING",   "STRING", "STRING",       "STRING", "DOUBLE", "DOUBLE_FUNCTION"]
          },
          "INSTANCE": [
            ["balance", "PITT", "balance_PITT",    "EQ",     450.0,    "DualValue(Row)"],
            ["balance", "NE",   "balance_NE",      "EQ",     0.0,      "DualValue(Row)"  ],
            ["balance", "SE",   "balance_SE",      "EQ",     0.0,      "DualValue(Row)"  ],
            ["balance", "BOS",  "balance_BOS",     "EQ",     -90.0,    "DualValue(Row)" ],
            ["balance", "EWR",  "balance_EWR",     "EQ",     -120.0,   "DualValue(Row)" ],
            ["balance", "BWI",  "balance_BWI",     "EQ",     -120.0,   "DualValue(Row)" ],
            ["balance", "ATL",  "balance_ATL",     "EQ",     -70.0,    "DualValue(Row)" ],
            ["balance", "MCO",  "balance_MCO",     "EQ",     -50.0 ,   "DualValue(Row)" ]
          ]
        },
        { "NAME": "totalCost",
          "CLASS": "OBJECTIVE",
          "KIND": "LINEAR",
          "SCHEMA": {
            "FIELDS":  ["Name",      "Row",         "Constant", "Sense",    "Value"],
            "TYPES":   ["STRING",    "STRING",      "DOUBLE",   "STRING",   "DOUBLE_FUNCTION"]
          },
          "INSTANCE": [["totalCost", "totalCost",    0.0,       "MINIMIZE", "ObjectiveValue(Row)"]]
        },
```

*Heading: Optional information for human readers not otherwise processed by MOSDEX*

*Math: Problem formulation in any algebraic language not otherwise processed by MOSDEX*

*Ship Table represents a decision variable artifact*

*Table Schema and Instance data*

*Outputs specified by Function Call fields*

*Figure 10: Full Transshipment Example in Instance Form (continued)*

```
          { "NAME": "balance_shipFrom",
            "CLASS": "TERM",
            "KIND": "LINEAR",
            "SCHEMA": {
              "FIELDS":
                ["Row",          "Column",       "Coefficient"],
              "TYPES":
                ["STRING",       "STRING",       "DOUBLE"]
            },
            "INSTANCE": [
              ["balance_PITT", "ship_PITT_NE", 1.0          ],
              ["balance_PITT", "ship_PITT_SE", 1.0          ],
              ["balance_NE",   "ship_NE_BOS",  1.0          ],
              ["balance_NE",   "ship_NE_EWR",  1.0          ],
              ["balance_NE",   "ship_NE_BWI",  1.0          ],
              ["balance_SE",   "ship_SE_EWR",  1.0          ],
              ["balance_SE",   "ship_SE_BWI",  1.0          ],
              ["balance_SE",   "ship_SE_ATL",  1.0          ],
              ["balance_SE",   "ship_SE_MCO",  1.0          ]
            ]
          },
          { "NAME": "balance_shipTo",
            "CLASS": "TERM",
            "KIND": "LINEAR",
            "SCHEMA": {
              "FIELDS":
                ["Row",         "Column",         "Coefficient"],
              "TYPES":
                ["STRING",      "STRING",         "DOUBLE"]
            },
            "INSTANCE": [
              ["balance_NE",  "ship_PITT_NE",    -1.0         ],
              ["balance_SE",  "ship_PITT_SE",    -1.0         ],
              ["balance_BOS", "ship_NE_BOS",     -1.0         ],
              ["balance_EWR", "ship_NE_EWR",     -1.0         ],
              ["balance_EWR", "ship_SE_EWR",     -1.0         ],
              ["balance_BWI", "ship_NE_BWI",     -1.0         ],
              ["balance_BWI", "ship_SE_BWI",     -1.0         ],
              ["balance_ATL", "ship_SE_ATL",     -1.0         ],
              ["balance_MCO", "ship_SE_MCO",     -1.0         ]
            ]
          },
          { "NAME": "total_ship",
            "CLASS": "TERM",
            "KIND": "LINEAR",
            "SCHEMA": {
              "FIELDS":
                ["Row",       "Column",       "Coefficient"],
              "TYPES":
                ["STRING",    "STRING",       "DOUBLE"]
            },
            "INSTANCE": [
              ["totalCost", "ship_PITT_NE", 2.5          ],
              ["totalCost", "ship_PITT_SE", 3.5          ],
              ["totalCost", "ship_NE_BOS",  1.7          ],
              ["totalCost", "ship_NE_EWR",  0.7          ],
              ["totalCost", "ship_NE_BWI",  1.3          ],
              ["totalCost", "ship_SE_EWR",  1.3          ],
              ["totalCost", "ship_SE_BWI",  0.8          ],
              ["totalCost", "ship_SE_ATL",  0.2          ],
              ["totalCost", "ship_SE_MCO",  2.1          ]
            ]
          }
        ]
      }
    ]
  }
```

**Appendix 2. Query-Form Example**

This appendix gives the full MOSDEX File for the **generalTransshipment** model in query form (Figure 11 through Figure 14).

**Lines 3-199:** Separate modules define the model, the data and the results.

**Lines 8-25**: The problem definition and Heading are the same as in Figure 11.

**Lines 27-43:** As in Figure 11, the Table **ship** represents the decision variables of the **generalTransshipment** problem. However, in Figure 12, this modeling artifact is defined by an SQL query rather by instance data.

**Lines 31-42:** A MOSDEX Query represents an SQL statement that specifies how the data are constructed from other Tables or from an external database. A Query consists of a list of *clauses*, each of which specifies a *directive*, which is a SQL command, and one or more *predicates*, the arguments of the directive, as the Query for the **ship** Variable illustrates. The **SELECT** clause's predicate specifies how the table's fields are accessed or computed from the fields of the parent table, **routes**. The column encoding is computed as the concatenation of the variable name and the two keys (again, MOSDEX does not specify the column encoding, and the modeler is free to choose any convenient encoding as a string or integer, provided there is no duplication). MOSDEX does not parse SQL, but the directives and predicates must be interpreted directly as valid SQL by the database engine. By tying MOSDEX to SQL, moreover, we take advantage of wide-spread expertise and computational systems available to support relational database systems.

**Lines 32-40:** The **ship** Variable table does not contain a Schema object, per se. Instead the schema is generated from the query. Specifically, the fields are named in the **AS** part of each item in the **SELECT** clause predicate; furthermore, their types are specified following the "--" symbol, which SQL reads as a comment.

**Lines 74-83:** The queries in this example illustrate the power of SQL to create compact but efficient data structures for optimization modeling. By joining tables, MOSDEX can use filtering, or slicing, to match data elements with variables and constraints. Such operations are typically computationally intensive and so are best performed using a database engine rather than hand-coded loops that are available in most programming languages.

**Lines 118-166:** The input data tables, **cities** and **routes**, are separated from the modeling artifacts. Since there is no external source for the data such as a database, they are presented as instance-form tables within the **generalTransshipment** problem; however, they also could be presented in a separate MOSDEX file. Furthermore, in the latter case, the same model could be invoked with completely different data. That is one advantage of using model/data separation.

**Lines 167-198:** The output data tables, **shipments** and **objective**, are separated from the modeling artifacts. SQL queries facilitate reshaping results for consumption by downstream applications.

Figure 4 shows the MOSDEX results file after a solve using IBM CPLEX. The solver result items are highlighted in green.

*Figure 11: Full Transshipment Example in Query Form*

```
{
  "SYNTAX": "MOSDEX/MOSDEX v2/MOSDEXSchemaV2-0.json",
  "MODULES": [
    {
      "NAME": "modelingObjects",
      "CLASS": "MODULE",
      "KIND": "MODEL",
      "HEADING": {
        "DESCRIPTION": [
          "General Transshipment Problem",
          "query form",
          "MOSDEX 2-0 Syntax"
        ],
        "VERSION": "net1a 2-1",
        "REFERENCE": ["https://ampl.com/BOOK/EXAMPLES/EXAMPLES2/net1.mod"],
        "AUTHOR": ["Jeremy A. Bloom (jeremyblmca@gmail.com)"],
        "NOTICES": ["Copyright 2019 Jeremy A. Bloom"],
        "MATH": [
          "var Ship {(i,j) in ROUTES} >= 0, <= capacity[i,j]; # packages to be shipped",
          "minimize Total_Cost: sum {(i,j) in ROUTES} cost[i,j] * Ship[i,j];",
          "subject to",
            "Balance {k in CITIES}: ",
              "sum {(k,j) in ROUTES} Ship[k,j] - sum {(i,k) in ROUTES} Ship[i,k] = supply[k] - demand[k];"
        ]
      },
      "TABLES": [
        {
           "NAME":"ship",
           "CLASS": "VARIABLE",
           "KIND": "CONTINUOUS",
          "QUERY": {
            "SELECT": [
              "'ship' AS Name                                        -- STRING",
              "routes.origin AS origin                               -- STRING",
              "routes.destination AS destination                     -- STRING",
              "CONCAT('ship', '_', origin, '_', destination) AS Column -- STRING",
              "CAST(0.0 AS DOUBLE) AS LowerBound                     -- DOUBLE",
              "routes.capacity AS UpperBound                         -- DOUBLE",
              "'PrimalValue(Column)' AS value                        -- DOUBLE_FUNCTION"
            ],
            "FROM": "routes"
          }
        },
        {
          "NAME": "balance",
          "CLASS": "CONSTRAINT",
          "KIND": "LINEAR",
          "QUERY": {
            "SELECT": [
              "'balance' AS Name                                     -- STRING",
              "cities.city AS city                                   -- STRING",
              "CONCAT('balance', '_', city) AS Row                   -- STRING",
              "'EQ' AS Sense                                         -- STRING",
              "(cities.supply-cities.demand) AS RHS                  -- DOUBLE",
              "'DualValue(Row)' AS dual                              -- DOUBLE_FUNCTION"
            ],
            "FROM": "cities"
          }
        },
        {
          "NAME": "totalCost",
          "CLASS": "OBJECTIVE",
          "KIND": "LINEAR",
          "SCHEMA": {
            "FIELDS": [  "Name",      "Row",       "Constant", "Sense",     "cost"],
            "TYPES": [   "STRING",    "STRING",    "DOUBLE",   "STRING",    "DOUBLE_FUNCTION"]
          },
          "INSTANCE": [[ "totalCost", "totalCost",  0.0,       "MINIMIZE",  "ObjectiveValue(Row)"]]
        },
```

*Modules separate model from data and results*

*Same Heading as Figure 2*

*Ship Table represents a decision variable artifact*

*SQL Query specifies how data are constructed from other Tables*

*MOSDEX schema is specified as part of the Select clause*

*TotalCost Table represents an objective function artifact Can be specified either as an instance or query*

*Figure 12: Full Transshipment Example in Query Form (continued)*

*Complex query illustrates joining two tables*

```
        {
          "NAME": "balance_shipFrom",
          "CLASS": "TERM",
          "KIND": "LINEAR",
          "QUERY": {
            "SELECT": [
              "balance.Row AS Row -- STRING",
              "ship.Column AS Column -- STRING",
              "CAST(1.0 AS DOUBLE) AS Coefficient -- DOUBLE"
            ],
            "FROM": "balance",
            "JOIN": "ship",
            "ON": "balance.city = ship.origin"
            }
        },
        {
          "NAME": "balance_shipTo",
          "CLASS": "TERM",
          "KIND": "LINEAR",
          "QUERY": {
            "SELECT": [
              "balance.Row AS Row -- STRING",
              "ship.Column AS Column -- STRING",
              "CAST(-1.0 AS DOUBLE) AS Coefficient -- DOUBLE"
            ],
            "FROM": "balance",
            "JOIN": "ship",
            "ON":  "balance.city = ship.destination"
          }
        },
        {
          "NAME": "total_ship",
          "CLASS": "TERM",
          "KIND": "LINEAR",
          "QUERY": {
            "SELECT": [
              "totalCost.Row AS Row -- STRING",
              "ship.Column AS Column -- STRING",
              "routes.cost AS Coefficient -- DOUBLE"
            ],
            "FROM": "totalCost",
            "CROSS JOIN": "ship",
            "JOIN": "routes",
            "ON": "routes.origin = ship.origin AND routes.destination = ship.destination"
          }
        }
      ]
    },
```

*Figure 13: Full Transshipment Example in Query Form (continued)*

```
    {
      "NAME": "data",
      "CLASS": "MODULE",
      "KIND": "DATA",
      "HEADING": {
        "DESCRIPTION": ["Data for General Transshipment Problem"]
      },
      "TABLES": [
        {
        "NAME": "cities",
        "CLASS": "DATA",
        "KIND": "INPUT",
        "SCHEMA": {
          "FIELDS": [ "city",   "supply",  "demand"],
          "TYPES": [  "STRING", "DOUBLE",  "DOUBLE"]
        },
        "INSTANCE": [
          [           "PITT",   450.0,     0.0    ],
          [           "NE",     0.0,       0.0    ],
          [           "SE",     0.0,       0.0    ],
          [           "BOS",    0.0,       90.0   ],
          [           "EWR",    0.0,       120.0  ],
          [           "BWI",    0.0,       120.0  ],
          [           "ATL",    0.0,       70.0   ],
          [           "MCO",    0.0,       50.0   ]
        ]
        },
        {
          "NAME": "routes",
          "CLASS": "DATA",
          "KIND": "INPUT",
          "SCHEMA": {
            "FIELDS": [ "origin",  "destination", "cost",   "capacity" ],
            "TYPES": [  "STRING",  "STRING",      "DOUBLE", "DOUBLE"]
          },
          "INSTANCE": [
            [           "PITT",    "NE",          2.5,     250.0     ],
            [           "PITT",    "SE",          3.5,     250.0     ],
            [           "NE",      "BOS",         1.7,     100.0     ],
            [           "NE",      "EWR",         0.7,     100.0     ],
            [           "NE",      "BWI",         1.3,     100.0     ],
            [           "SE",      "EWR",         1.3,     100.0     ],
            [           "SE",      "BWI",         0.8,     100.0     ],
            [           "SE",      "ATL",         0.2,     100.0     ],
            [           "SE",      "MCO",         2.1,     100.0     ]
          ]
        }
      ]
    },
```

*Data Module*

*Cities Table represents an input data object*

*Table Schema and Instance data*

*Figure 14: Full Transshipment Example in Query Form (continued)*

```
    {
      "NAME": "results",
      "CLASS": "MODULE",
      "KIND": "DATA",
      "HEADING": {
        "DESCRIPTION": ["Results from General Transshipment Problem"]
      },
      "TABLES": [
        {
          "NAME": "shipments",
          "CLASS": "DATA",
          "KIND": "OUTPUT",
          "QUERY": {
            "SELECT": [
              "ship.origin AS origin           -- STRING",
              "ship.destination AS destination -- STRING",
              "ship.value AS value             -- DOUBLE"
            ],
            "FROM": "ship"
          }
        },
        {
          "NAME": "objective",
          "CLASS": "DATA",
          "KIND": "OUTPUT",
          "QUERY": {
            "SELECT": ["totalCost.cost AS cost -- DOUBLE"],
            "FROM": "totalCost"
          }
        }
      ]
    },
  ]
}
```

*Results Module*

*Shipments Table represents an output data object*
*SQL Query facilitates reshaping results for consumption by downstream applications*

*Figure 15: Results after CPLEX Solve*

*Results Module*

*Solver outputs from CPLEX solve*


```
{
  "NAME" : "results",
  "KIND" : "MODULE",
  "HEADING" : {
    "DESCRIPTION" : [ "Results from General Transshipment Problem" ]
  },
  "TABLES" : [
    {
    "NAME" : "shipments",
    "CLASS" : "DATA",
    "KIND" : "OUTPUT",
    "SCHEMA" : {
      "FIELDS" : [ "origin", "destination", "value" ],
      "TYPES" : [ "STRING", "STRING", "DOUBLE" ] },
    "INSTANCE" : [
       [ "PITT", "NE", 250.0 ],
       [ "PITT", "SE", 200.0 ],
       [ "NE", "BOS", 90.0 ],
       [ "NE", "EWR", 100.0 ],
       [ "NE", "BWI", 60.0 ],
       [ "SE", "EWR", 20.0 ],
       [ "SE", "BWI", 60.0 ],
       [ "SE", "ATL", 70.0 ],
       [ "SE", "MCO", 50.0 ] ]
    },
    {
    "NAME" : "objective",
    "CLASS" : "DATA",
    "KIND" : "OUTPUT",
    "SCHEMA" : {
      "FIELDS" : [ "cost" ],
      "TYPES" : [ "DOUBLE" ] },
    "INSTANCE" : [ [ 1819.0 ] ]
    } ]
  } ]
}
```


**MOSDEX Resources**

[i] MOSDEX Schema: https://github.com/JeremyBloom/MOSDEX-Reference-Architecture/blob/main/MOSDEXSchemaV2-0.json

[ii] Simple linear program example: https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/volsay_2-0.json

[iii] Network example (instance form): https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/net1b_2-1.json

[iv] Network example (query form): https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/net1a_2-1.json

[v] LP example with lagged variables: https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/sailco_2-0.json

[vi] Warehouse location example (MIP): https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/warehousing_2-0.json

[vii] Traffic Network QP: https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-2.0/trafficNetworkQP_2-0.json

[viii] Warehouse Location with Stochastic Demand: https://github.com/JeremyBloom/Optimization---Sample-Notebooks/blob/master/Locating%2BWarehouses%2Bto%2BMinimize%2BCosts%2BCase%2B2%2Bpub%2B3.html

[ix] MOSDEX Reference Architecture: https://github.com/JeremyBloom/MOSDEX-Reference-Architecture