**MOSDEX: A New Standard for Data Exchange with Optimization Solvers**

Dr. Jeremy A. Bloom  
[jeremyblmca@gmail.com](mailto:jeremyblmca@gmail.com)  
November 3, 2019

Other co-authors

**Abstract**

MPS format, which has become the standard for data exchange with mathematical optimization solvers, has serious deficiencies when it comes to supporting modern decision applications based on optimization. This paper proposes a new standard called MOSDEX to overcome them. 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 objects. 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 well-known optimization problems. 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**

The MPS format “has emerged as a de facto standard ASCII medium among most of the commercial LP solvers.” (Wikipedia, 2018). Although its usage has declined with the widespread adoption of algebraic modeling languages such as OPL, AMPL, and GAMS, the need for a non-proprietary standard that can support multiple solvers remains strong. However, the limitations of the MPS format are becoming increasingly apparent, and therefore in this document, we want to propose an alternative called **MOSDEX** (**M**athematical **O**ptimization **S**olver **D**ata **Ex**change), which hopefully can become a new standard to supersede the older MPS standard.

Before addressing the deficiencies we hope to remedy with an alternative, let us first consider the advantages that MPS brings:

* 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 reductions 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 suffers from a number of shortcomings. Among these, its fixed-column format, while archaic, is perhaps the least important, since that aspect is rather easily overcome and many solvers that accept MPS files that do not enforce it. More important are the following:

1. **Lack of an output standard**: MPS is aformat for input and there exists no corresponding standard for output from an optimization solver.
2. **Lack of model-data separation**: This aspect means that the model (variables, constraints, objective) is intertwined with the data that populates it rather than specified independently of the data populating any particular instance. It is considered a best practice to separate them, for a number of reasons discussed below. Furthermore, the widely used modeling languages are designed for model-data separation.
3. **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 instance a distribution network, does not change but the data represents different numbers of entities, such as warehouses, stores, and routes between them.
4. **Lack of indexing**: One way to achieve scalability is to 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 *x1, x2, x3*. In MPS, 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 object (although the use of columns to specify matrix data would also need to be revised as discussed in the next item).
5. **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.) 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).
6. **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 replacement for MPS needs to support these extensions in a standard way.

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

1. **Efficient for machines, readable by humans**: 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, for instance for debugging and documentation purposes.
2. **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.
3. **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 discuss below the reasons for choosing JSON over XML, the other widely adopted data format standard.
4. **When necessary, augment the data representation with mathematical modeling objects (variables, constraints, objectives, etc.) 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 objects. 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. However, MOSDEX 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.)
5. **MOSDEX Overview**

At the highest level, a *MOSDEX File* is a collection of optimization *Problems*, each of which in turn is a collection of *Tables*. A Table represents a table in a relational database. Subclasses of Table represent Data and modeling objects, such as Variables and Constraints. 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 objects are largely fixed by the requirements of an optimization solver. A Table can have either *instance* or *recipe* form. Instance form tables contain data while recipe form tables use SQL queries to reshape and populate data from other tables. A MODEX problem can contain both instance and recipe form tables, although using recipe form tables presupposes the presence of a database in the software stack that executes the optimization application.

The figures 1-3 below illustrate specification of a simple linear program in MOSDEX. It represents a transshipment network, and it is borrowed from <https://ampl.com/BOOK/EXAMPLES/EXAMPLES2/net1.mod>. This example uses purely instance form tables for the modeling objects and simple recipe tables for output (the meaning of “simple” in this context and the reasons for using recipes for output are discussed below). This particular example is illustrative of how MOSDEX compares with MPS.

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 <http://json.org/>). Second, MOSDEX is specified using a standard JSON Schema (see <http://json-schema.org/>). 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 and FIELDS 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 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 either an *identifier* or a *keyword*; the latter is denoted by using all capital letters.

A JSON array 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, 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 use this format, it provides the most precise way to exchange numerical data with a solver.

Figure 1: Transshipment Example in Instance Form

**5**

**4**

**3**

**2**

**1**

{  
 **"SYNTAX"**: **"MOSDEX/MOSDEX v1-2/MOSDEXSchemaV1-2.json"**,  
 **"generalTransshipment"**: {  
 **"CLASS"**: **"PROBLEM"**,  
 **"HEADING"**: {  
 **"DESCRIPTION"**: [  
 **"General Transshipment Problem"**,  
 **"instance form"**, **"with a simple recipe for output"**,  
 **"MOSDEX 1-2 Syntax"** ],  
 **"VERSION"**: [**"b 1-2-2"**],  
 **"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 LINKS} >= 0, <= capacity[i,j]; # packages to be shipped"**,  
 **"minimize Total\_Cost: sum {(i,j) in LINKS} cost[i,j] \* Ship[i,j];"**,  
 **"subject to"**,  
 **"Balance {k in CITIES}: "**,  
 **"sum {(k,j) in LINKS} Ship[k,j] - sum {(i,k) in LINKS} Ship[i,k] = supply[k] - demand[k];"** ]  
 },  
 **"ship"**: {  
 **"CLASS"**: **"VARIABLE"**,  
 **"TYPE"**: **"CONTINUOUS"**,  
 **"SCHEMA"**: {  
 **"Name"**: **"STRING"**,  
 **"origin"**: **"STRING"**,  
 **"destination"**: **"STRING"**,  
 **"Column"**: **"STRING"**,  
 **"Type"**: **"STRING"**,  
 **"LowerBound"**: **"DOUBLE"**,  
 **"UpperBound"**: **"DOUBLE"**,  
 **"Value"**: **"DOUBLE"** },  
 **"FIELDS"**: [  
 **"Name"**, **"origin"**, **"destination"**, **"Column"**, **"Type"**, **"LowerBound"**, **"UpperBound"**, **"Value"**],  
 **"INSTANCE"**: [  
 [**"ship"**, **"PITT"**, **"NE"**, **"ship\_PITT\_NE"**, **"CONTINUOUS"**, 0.0, 250.0, **null** ],  
 [**"ship"**, **"PITT"**, **"SE"**, **"ship\_PITT\_SE"**, **"CONTINUOUS"**, 0.0, 250.0, **null** ],  
 [**"ship"**, **"NE"**, **"BOS"**, **"ship\_NE\_BOS"**, **"CONTINUOUS"**, 0.0, 100.0, **null** ],  
 [**"ship"**, **"NE"**, **"EWR"**, **"ship\_NE\_EWR"**, **"CONTINUOUS"**, 0.0, 100.0, **null** ],  
 [**"ship"**, **"NE"**, **"BWI"**, **"ship\_NE\_BWI"**, **"CONTINUOUS"**, 0.0, 100.0, **null** ],  
 [**"ship"**, **"SE"**, **"EWR"**, **"ship\_SE\_EWR"**, **"CONTINUOUS"**, 0.0, 100.0, **null** ],  
 [**"ship"**, **"SE"**, **"BWI"**, **"ship\_SE\_BWI"**, **"CONTINUOUS"**, 0.0, 100.0, **null** ],  
 [**"ship"**, **"SE"**, **"ATL"**, **"ship\_SE\_ATL"**, **"CONTINUOUS"**, 0.0, 100.0, **null** ],  
 [**"ship"**, **"SE"**, **"MCO"**, **"ship\_SE\_MCO"**, **"CONTINUOUS"**, 0.0, 100.0, **null** ]  
 ]  
 },

Figure 2: Transshipment Example in Instance Form (continued)

**8**

**7**

**6**

**"balance"**: {  
 **"CLASS"**: **"CONSTRAINT"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"SCHEMA"**: {  
 **"Name"**: **"STRING"**,  
 **"city"**: **"STRING"**,  
 **"Row"**: **"STRING"**,  
 **"Type"**: **"STRING"**,  
 **"Sense"**: **"STRING"**,  
 **"RHS"**: **"DOUBLE"** },  
 **"FIELDS"**: [  
 **"Name"**, **"city"**, **"Row"**, **"Type"**, **"Sense"**, **"RHS"** ],  
 **"INSTANCE"**: [  
 [**"balance"**, **"PITT"**, **"balance\_PITT"**, **"LINEAR"**, **"EQ"**, 450.0 ],  
 [**"balance"**, **"NE"**, **"balance\_NE"**, **"LINEAR"**, **"EQ"**, 0.0 ],  
 [**"balance"**, **"SE"**, **"balance\_SE"**, **"LINEAR"**, **"EQ"**, 0.0 ],  
 [**"balance"**, **"BOS"**, **"balance\_BOS"**, **"LINEAR"**, **"EQ"**, -90.0 ],  
 [**"balance"**, **"EWR"**, **"balance\_EWR"**, **"LINEAR"**, **"EQ"**, -120.0 ],  
 [**"balance"**, **"BWI"**, **"balance\_BWI"**, **"LINEAR"**, **"EQ"**, -120.0 ],  
 [**"balance"**, **"ATL"**, **"balance\_ATL"**, **"LINEAR"**, **"EQ"**, -70.0 ],  
 [**"balance"**, **"MCO"**, **"balance\_MCO"**, **"LINEAR"**, **"EQ"**, -50.0 ]  
 ]  
 },  
 **"totalCost"**: {  
 **"CLASS"**: **"OBJECTIVE"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"SINGLETON"**: {  
 **"Name"**: **"totalCost"**,  
 **"Row"**: **"totalCost"**,  
 **"Type"**: **"LINEAR"**,  
 **"Constant"**: 0.0,  
 **"Sense"**: **"MINIMIZE"**,  
 **"Value"**: **null** }  
 },  
 **"balance\_shipFrom"**: {  
 **"CLASS"**: **"TERM"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"SCHEMA"**: {  
 **"Type"**: **"STRING"**,  
 **"Row"**: **"STRING"**,  
 **"Column"**: **"STRING"**,  
 **"Coefficient"**: **"DOUBLE"** },  
 **"FIELDS"**:  
 [**"Type"**, **"Row"**, **"Column"**, **"Coefficient"**],  
 **"INSTANCE"**: [  
 [**"LINEAR"**, **"balance\_PITT"**, **"ship\_PITT\_NE"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_PITT"**, **"ship\_PITT\_SE"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_NE"**, **"ship\_NE\_BOS"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_NE"**, **"ship\_NE\_EWR"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_NE"**, **"ship\_NE\_BWI"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_SE"**, **"ship\_SE\_EWR"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_SE"**, **"ship\_SE\_BWI"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_SE"**, **"ship\_SE\_ATL"**, 1.0 ],  
 [**"LINEAR"**, **"balance\_SE"**, **"ship\_SE\_MCO"**, 1.0 ]  
 ]  
 },

Figure 3: Transshipment Example in Instance Form (continued)

**9**

**"balance\_shipTo"**: {  
 **"CLASS"**: **"TERM"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"SCHEMA"**: {  
 **"Type"**: **"STRING"**,  
 **"Row"**: **"STRING"**,  
 **"Column"**: **"STRING"**,  
 **"Coefficient"**: **"DOUBLE"** },  
 **"FIELDS"**:  
 [**"Type"**, **"Row"**, **"Column"**, **"Coefficient"**],  
 **"INSTANCE"**: [  
 [**"LINEAR"**, **"balance\_NE"**, **"ship\_PITT\_NE"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_SE"**, **"ship\_PITT\_SE"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_BOS"**, **"ship\_NE\_BOS"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_EWR"**, **"ship\_NE\_EWR"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_EWR"**, **"ship\_SE\_EWR"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_BWI"**, **"ship\_NE\_BWI"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_BWI"**, **"ship\_SE\_BWI"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_ATL"**, **"ship\_SE\_ATL"**, -1.0 ],  
 [**"LINEAR"**, **"balance\_MCO"**, **"ship\_SE\_MCO"**, -1.0 ]  
 ]  
 },  
 **"total\_ship"**: {  
 **"CLASS"**: **"TERM"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"SCHEMA"**: {  
 **"Type"**: **"STRING"**,  
 **"Row"**: **"STRING"**,  
 **"Column"**: **"STRING"**,  
 **"Coefficient"**: **"DOUBLE"** },  
 **"FIELDS"**:  
 [**"Type"**, **"Row"**, **"Column"**, **"Coefficient"**],  
 **"INSTANCE"**: [  
 [**"LINEAR"**, **"totalCost"**, **"ship\_PITT\_NE"**, 2.5 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_PITT\_SE"**, 3.5 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_NE\_BOS"**, 1.7 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_NE\_EWR"**, 0.7 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_NE\_BWI"**, 1.3 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_SE\_EWR"**, 1.3 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_SE\_BWI"**, 0.8 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_SE\_ATL"**, 0.2 ],  
 [**"LINEAR"**, **"totalCost"**, **"ship\_SE\_MCO"**, 2.1 ]  
 ]  
 },  
 **"shipments"**: {  
 **"CLASS"**: **"DATA"**,  
 **"TYPE"**: **"OUTPUT"**,  
 **"RECIPE"**: [  
 {**"DIRECTIVE"**: [**"SELECT"**], **"PREDICATE"**: [**"origin"**, **"destination"**, **"#Solver.getValue('Column')"**, **"AS shipment"**]},  
 {**"DIRECTIVE"**: [**"FROM"**], **"PREDICATE"**: [**"ship"**]}  
 ]  
 },  
 **"objective"**: {  
 **"CLASS"**: **"DATA"**,  
 **"TYPE"**: **"OUTPUT"**,  
 **"RECIPE"**: [  
 {**"DIRECTIVE"**: [**"SELECT"**], **"PREDICATE"**: [**"#Solver.getObjValue()"**,**"AS TotalCost"**]},  
 {**"DIRECTIVE"**: [**"FROM"**], **"PREDICATE"**: [**"totalCost"**]}  
 ]  
 }  
 }  
}

Let’s walk through this example step-by-step:

1. A MOSDEX *File* consists of one of more *Problems*. (We use the term file generically for any input source.) The example has just one, a named object called **generalTransshipment**. It is useful to think of a Problem as a self-contained presentation of the data and modeling objects 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 Problems without modeling objects.
2. A *Heading* element, a keyword object, provides documentation for human readers. A Heading is required for a Problem, 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.
3. The main elements of the Problem are its *Tables*, contained in one or more named objects. 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 objects, *Variable*, *Constraint*, *Objective*, and *Term*, are subclasses of Table. A Table’s Class and Type 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.
4. 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 format for this example (a concatenation of the variable name and 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.
5. The Instance object of the **ship** Variable contains actual data. In an *Instance* object, the individual items in each *Record* are unlabeled and can only be parsed using the Table’s Schema. The Instance may be preceded by an optional *Fields* array, which repeats the schema’s field names as a visual guide for a human reader. Notice that, in contrast to MPS, MOSDEX does not include coefficient data among the data specifying a Variable. Instead, coefficients are specified in separate Terms objects, discussed in item 9 below.
6. 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.
7. The Table **totalCost** represents the objective function of the **generalTransshipment** Problem. This table has a single row, so we use a *Singleton* object, which is a subtype of Instance that provides a simplified, compact representation of such Tables. In a Singleton object, the field labels constitute the field names of the Singleton’s schema, and the field types (which are limited to JSON Primitives) are inferred by the parser. Note that the Objective is assigned a Row identifier.
8. The Term object **balance\_shipFrom** represents the first of several tables that specify the coefficients of the **generalTransshipment** Problem. The name of a Term object 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 object identifies the Row and Column to which each coefficient applies. Note that Term objects 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.
9. Outputs in MOSDEX are specified by simple recipes, even when the rest of the model is specified by instance-form tables. A *Recipe* 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. The field denoted **#Solver.getValue(Column)** is a function call that gets data from the solver. Recipes are discussed in more detail in the next section. The reason for using a Recipe for output is that the output Tables cannot be populated until after the solver has run, so it is meaningless to create them in the preprocessing phase, unlike the input Tables and modeling objects. Without a SQL engine, however, there are limits on the kinds of recipes that can be implemented; essentially, they are limited to selecting fields from a single Table. Such simple Recipes can be implemented by iterating over the Records of the Table. However, joins of two or more Tables are prohibited in simple recipes.

This concludes discussion of the instance-form example of MOSDEX.

1. **Recipe Form Tables and the Relational Data Model**

The instance form Tables of MOSDEX shown in the example of the previous section probably seem familiar to many, especially those who have used MPS, 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 moves through a pipeline from the data store to the optimization application. This process is a critical part of designing and operating an optimization application, often entailing substantial software development and computational effort.

Mathematical optimization solvers 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 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 pipeline, 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 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 pipeline. 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 (Bloom, 2017), there is a deep relationship between the structure of the data used in optimization modeling and the constructs (variables, constraints) 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 constructions 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 *recipe* *form*. The following example, figures 4-7, shows the same transshipment example of section 2 but in recipe form:

Figure 4: Transshipment Example in Recipe Form

**2**

**1**

{  
 **"SYNTAX"**: **"MOSDEX/MOSDEX v1-2/MOSDEXSchemaV1-2.json"**,  
 **"generalTransshipment"**: {  
 **"CLASS"**: **"PROBLEM"**,  
 **"HEADING"**: {  
 **"DESCRIPTION"**: [  
 **"General Transshipment Problem"**,  
 **"recipe form"**,  
 **"MOSDEX 1-2 Syntax"** ],  
 **"VERSION"**: [**"a 1-2"**],  
 **"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 LINKS} >= 0, <= capacity[i,j]; # packages to be shipped"**,  
 **"minimize Total\_Cost: sum {(i,j) in LINKS} cost[i,j] \* Ship[i,j];"**,  
 **"subject to"**,  
 **"Balance {k in CITIES}: "**,  
 **"sum {(k,j) in LINKS} Ship[k,j] - sum {(i,k) in LINKS} Ship[i,k] = supply[k] - demand[k];"** ]  
 },  
 **"cities"**: {  
 **"CLASS"**: **"DATA"**,  
 **"TYPE"**: **"INPUT"**,  
 **"SCHEMA"**: {  
 **"city"**: **"STRING"**,  
 **"supply"**: **"DOUBLE"**,  
 **"demand"**: **"DOUBLE"** },  
 **"FIELDS"**: [**"city"**, **"supply"**, **"demand"**],  
 **"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 ]  
 ]  
 },

Figure 5: Transshipment Example in Recipe Form (continued)

**3**

**"links"**: {  
 **"CLASS"**: **"DATA"**,  
 **"TYPE"**: **"INPUT"**,  
 **"SCHEMA"**: {  
 **"origin"**: **"STRING"**,  
 **"destination"**: **"STRING"**,  
 **"cost"**: **"DOUBLE"**,  
 **"capacity"**: **"DOUBLE"** },  
 **"FIELDS"**: [**"origin"**, **"destination"**, **"cost"**, **"capacity"** ],  
 **"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 ]  
 ]  
},  
**"ship"**: {  
 **"CLASS"**: **"VARIABLE"**,  
 **"TYPE"**: **"CONTINUOUS"**,  
 **"RECIPE"**:[  
 {**"DIRECTIVE"**: [**"QUERY"**], **"PREDICATE"**: [  
 **"SELECT"**,  
 **"'ship' AS Name"**,  
 **"'CONTINUOUS' AS Type"**,  
 **"origin"**, **"destination"**,  
 **"CONCAT('ship','\_', origin,'\_',destination) AS Column "**,  
 **"0.0 AS LowerBound"**,  
 **"capacity AS UpperBound"**,  
 **"NULL AS Value"**,  
 **"FROM"**, **"links"**]  
 }  
 ]  
},  
**"balance"**: {  
 **"CLASS"**: **"CONSTRAINT"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"RECIPE"**:[  
 {**"DIRECTIVE"**: [**"QUERY"**], **"PREDICATE"**: [  
 **"SELECT"**,  
 **"'balance' AS Name"**,  
 **"'LINEAR' AS Type"**,  
 **"city"**,  
 **"CONCAT('balance','\_',city) AS Row"**,  
 **"'EQ' AS Sense"**,  
 **"(supply-demand) AS RHS"**,  
 **"FROM"**, **"cities"**]  
 }  
 ]  
},

Figure 6: Transshipment Example in Recipe Form (continued)

**5**

**4**

**"totalCost"**: {  
 **"CLASS"**: **"OBJECTIVE"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"SINGLETON"**: {  
 **"Name"**:**"totalCost"**,  
 **"Row"**: **"totalCost"**,  
 **"Type"**: **"LINEAR"**,  
 **"Constant"**: 0.0,  
 **"Sense"**: **"MINIMIZE"**,  
 **"Value"**: **null** }  
},  
**"balance\_shipFrom"**: {  
 **"CLASS"**: **"TERM"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"RECIPE"**: [  
 {**"DIRECTIVE"**: [**"SELECT"**], **"PREDICATE"**: [  
 **"'LINEAR' AS Type"**,  
 **"balance.Row AS Row"**,  
 **"ship.Column AS Column"**,  
 **"1.0 AS Coefficient"**]  
 },  
 {**"DIRECTIVE"**: [**"FROM"**], **"PREDICATE"**: [**"balance"**]},  
 {**"DIRECTIVE"**: [**"JOIN"**], **"PREDICATE"**: [**"ship"**]},  
 {**"DIRECTIVE"**: [**"ON"**], **"PREDICATE"**: [ **"(balance.city == ship.origin)"**]}  
 ]  
},

**"balance\_shipTo"**: {  
 **"CLASS"**: **"TERM"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"RECIPE"**: [  
 {**"DIRECTIVE"**: [**"SELECT"**], **"PREDICATE"**: [  
 **"'LINEAR' AS Type"** ,  
 **"balance.Row AS Row"**,  
 **"ship.Column AS Column"**,  
 **"-1.0 AS Coefficient"**]  
 },  
 {**"DIRECTIVE"**: [**"FROM"**], **"PREDICATE"**: [**"balance"**]},  
 {**"DIRECTIVE"**: [**"JOIN"**], **"PREDICATE"**: [**"ship"**]},  
 {**"DIRECTIVE"**: [**"ON"**], **"PREDICATE"**: [**"(balance.city == ship.destination)"**]}  
 ]  
},  
**"total\_ship"**: {  
 **"CLASS"**: **"TERM"**,  
 **"TYPE"**: **"LINEAR"**,  
 **"RECIPE"**: [  
 {**"DIRECTIVE"**: [**"SELECT"**], **"PREDICATE"**: [  
 **"'LINEAR' AS Type"**,  
 **"totalCost.Row AS Row"**,  
 **"ship.Column AS Column"**,  
 **"links.cost AS Coefficient"**]  
 },  
 {**"DIRECTIVE"**: [**"FROM"**], **"PREDICATE"**: [**"totalCost"**]},  
 {**"DIRECTIVE"**: [**"JOIN"**], **"PREDICATE"**: [**"ship"**]},  
 {**"DIRECTIVE"**: [**"JOIN"**], **"PREDICATE"**: [**" links"**]},  
 {**"DIRECTIVE"**: [**"ON"**], **"PREDICATE"**: [**"(links.origin == ship.origin)"**, **"AND"**,

**"(links.destination == ship.destination)"**]}  
 ]  
},

Figure 7: Transshipment Example in Recipe Form (continued)

**"shipments"**: {  
 **"CLASS"**: **"DATA"**,  
 **"TYPE"**: **"OUTPUT"**,  
 **"RECIPE"**: [  
 {**"DIRECTIVE"**: [**"SELECT"**], **"PREDICATE"**: [**"origin"**, **"destination"**, **"Value AS shipment"**]},  
 {**"DIRECTIVE"**: [**"FROM"**], **"PREDICATE"**: [**"ship"**]}  
 ]  
 },  
 **"objective"**: {  
 **"CLASS"**: **"DATA"**,  
 **"TYPE"**: **"OUTPUT"**,  
 **"RECIPE"**: [  
 {**"DIRECTIVE"**: [**"SELECT"**], **"PREDICATE"**: [**"Value AS TotalCost"**]},  
 {**"DIRECTIVE"**: [**"FROM"**], **"PREDICATE"**: [**"totalCost"**]}  
 ]  
 }  
}

Let’s walk through this example step-by-step:

1. The Problem definition and Heading are the same as in figure 1.
2. The input data tables, **cities** and **links**, are now separated from the modeling objects. 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 problem or even in a separate file. Furthermore, in the latter case, the same model could be invoked with completely different data. That is one advantage of using data/model separation.
3. A Recipe is an SQL statement that specifies how the data are constructed from other Tables or from an external database. A recipe 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. The Recipes for the **ship** Variable and the **balance** Constraint illustrate the simplest form of Recipe, a single text string of SQL in the Predicate, preceded by the Directive **"QUERY"**. MOSDEX does not parse SQL, so it affords considerable flexibility in how the SQL is written, as long as the directive and predicate can 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 database systems.

Note that the **ship** Variable does not have a Schema object, per se. Instead the schema is generated by the Recipe. Specifically, the fields are named in the predicate of the Select clause, which also specifies how those fields are accessed or computed from the fields of the parent table, **links**. 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).

1. The Recipes for the **balance\_shipFrom** and **balance\_shipTo** Terms illustrate more verbose query specifications. In these queries, the Clauses represent separate components of an SQL statement. MOSDEX permits this more verbose syntax to enable customized parsers that need to identify the individual Clauses. For instance, not all database management systems natively support SQL of any flavor (e.g. Python Pandas), so using a Recipe object in MOSDEX that allows breaking a query into its component clauses enables adapting to such systems.
2. This recipe illustrates 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.

This concludes discussion of the recipe-form example of MOSDEX.

1. **Additional Capabilities of MOSDEX**

Modern optimization solvers support a variety of problem types that were not generally available when MPS was invented. As a result, MPS has been extended to support these additional capabilities, sometimes with solver-specific syntax that is 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:

**4.1 Linear, Mixed Integer, and Quadratic Problems**

The Type element of a Variable can be designated as Continuous, Integer, or Binary. The Type element of a Constraint, Objective, or 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. Although not implemented yet, MOSDEX is extensible to special structures such as special ordered sets, indicator variables, and so on. Here are some examples using these capabilities:

* Warehouse location (MIP) – <https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-1.2/warehousing_1-2.json>
* Traffic Network (QP) – <https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-1.2/trafficNetworkQP_1-2.json>

**4.2 Modular Structures**

Modular structures arise when two or more Problems 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. Here are some examples:

* Cutting Stock (column generation) – <https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-1.2/cuttingStock_1-2.json>
* ??? (stochastic program) – TBD

MOSDEX provides several constructs that enable building modular structures. First, a Table can be Imported from, Initialized from, or Exported to another Problem.

Second, MOSDEX provides a limited capability to use string variables. In MOSDEX, a string beginning with a $ sign denotes a string variable. The parser will substitute a string variable with an actual string as it processes a MOSDEX file. MOSDEX permits string variables in two situations:

1. To substitute a path string for a path name in a Table reference, which usually occurs in an Import, Initialization, or Export or in the From or Join clause of an SQL query.

2. To substitute an actual value for an index variable (discussed below), which usually occurs in a Select, Where, or On clause of an SQL query.

Third, MOSDEX provides a capability to use Index variables. It is frequently the case that the parent problem will want to incorporate multiple versions of a child problem. For example, in a multi-commodity flow problem, there could be multiple versions of a minimum cost flow problem, one for each product. Index variables deal with this common situation. An index variable is bound to a Table and represents the records as its index set. An index variable is called by For Each objects in other Tables, which, in effect, creates a loop in which the host Table is reproduced for each different index value, that is, for each record in the index set corresponding to the index variable. Usually the reproduced tables will be collected by a Union or Outer Join clause in an SQL query, since MOSDEX does not support arrays of Tables.

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.

**4.3 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 expressions. The MOSDEX representation is based on the JSON ADFun proposal by Brad Bell at <https://coin-or.github.io/CppAD/doc/json_ad_graph.htm> and <https://coin-or.github.io/CppAD/doc/to_json.cpp.htm>.

Here is an example:

* Traffic Network (QP as NLP) - <https://github.com/coin-modeling-dev/MOSDEX-Examples/blob/master/MOSDEX-1.2/trafficNetworkNLP_1-2.json>

To support extension to nonlinear problems, MOSDEX provides an Expression class, a subclass of Table, which embodies a node of an expression graph. Expressions themselves behave like other Tables (i.e. they each have a defined schema, and they can have Instance or Recipe form). The main innovation in MOSDEX is that an Expression has dimensionality as specified by its key fields. Thus MOSDEX implements families of Expressions that correspond to the structure of the optimization problem that it represents.

Expression, like the other modeling objects, is a subclass of Table in MOSDEX; as named JSON Objects, multiple Expression objects can coexist within a MOSDEX Problem. Typically, a linked collection of Expressions comprises a nonlinear expression in the Problem; multiple collections of Expressions can represent different nonlinear expressions in the Problem. Each Expression is one of 4 types: Independent Variable, Dependent Variable, Parameter, or operation. Independent and Dependent Variables link to a Variable modeling object; the distinction between them is that Independent Variables are inputs to the expression and Dependent Variables are outputs. Parameters link to data items and do not interact with the solver. Finally, operations represent operators that combine Independent Variables, other Expressions, and Parameters; typically they represent elementary math operations such as add or multiply, or math functions, such as *exp* or *log*. Expressions combine in a graph-like fashion that enables building complex expressions from elementary operations. The use of expression graphs is common to representing nonlinear optimization problems, and it facilitates automatic differentiation used by nonlinear solvers.

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.

1. **Architectural Considerations**

To Be written

1. **Conclusion and Next Steps**

This paper has proposed a new standard called MOSDEX to overcome deficiencies in the MPS standard 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 objects. 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.

The following steps are suggested for further development of MOSDEX:

1. To be written

**References**

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

Data Mining Group. (n.d.). *Predictive Model Markup Language.* Retrieved from http://dmg.org/

Faster:XML. (n.d.). *FasterXML.* Retrieved from http://fasterxml.com/

Fourer, R., Ma, J., and Martin, K. (2010). OSiL: An Instance Language for Optimization. *Comput. Optim. Appl., 45*(1), 181–203.

Gassmann, H., Ma, J., and Martin, K. (2016). Communication protocols for options and results. *Math. Prog. Comp., 8*, 161–189.

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

**About the Author**

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