**Architectural Considerations for MOSDEX**

MOSDEX files serve as repositories for the data (and modeling objects, 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, or *spans*, of the workflow bridges for an optimization-based application are as follows:

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 optimization.
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 solver calls to sophisticated object-oriented class libraries.
4. The solver executes its optimization algorithm, and the results are exposed through its API.
5. Data extracted 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 human users.

In the first two spans, MOSDEX serves as the format that enables standardization. MOSDEX’s use of recipe-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 span are captured in internal objects of some kind of business application platform built on top of a programming language like Java or Python. 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 span transforms the MODEX Object Model into classes of the solver’s API. In this step, MOSDEX serves to standardize the object model.

The fourth and fifth spans reverse the three input spans 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 spans, the MOSDEX standard serves to document the execution of these transformations.

As noted above, the data processing 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 in internet search and machine learning, which resulted in development of the software libraries Hadoop and Spark, among others. They provide the capability to distribute processing across clusters of computers, operating in parallel.

Spark, in particular, uses an abstraction for a bridge called a *resilient distributed dataset* (RDD), which partitions data items and distributes them onto the nodes of a cluster and which enables certain span operations, 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 *action*, requires returning a result to the calling program, or *driver*. The queue can then be reordered to optimize computing all of the transformations. The prototype action is *reduce*, which aggregates the data items according to some function, such as a sum, and returns a final result to the driver. An 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 objects 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.