Algebraic Modeling in Datalog

C. Borraz-Sánchez[†], J. Ma, D. Klabjan
[†]Data&Analytics Center, KPMG LLP, Knoxville, TN 37919
cborrazsanchez@kpmg.com

Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL 60208
{c-borraz-sanchez, maj, d-klabjan}@northwestern.edu

E. Pašalić, M. Aref
LogicBlox Inc., Atlanta, GA.
{emir.pasalic,molham.aref}@logicblox.com

Abstract: Datalog is a deductive language tailored for easy database access. We introduce an algebraic modeling language in Datalog for mixed-integer linear optimization models. By using this language, data can be easily queried from a database by means of Datalog and combined with models to produce problem instances readily available to solvers, providing an advantage over conventional optimization modeling languages that rely on reading data via plug-in tools or importing data from external sources via standard files. The declarative nature of Datalog permits the underlying syntax to be kept intuitive and simple. In this paper, we present the algebraic and syntactic ideas behind our proposed platform. Two specific case studies, a production-transportation and traveling salesman model, are presented. A comparison analysis between GAMS, a commercially available algebraic modeling system, and Datalog's algebraic modeling embedded system is also conducted. Datalog's main advantages are clearly exposed when solving real-life decision making problems embedding flexible business rules.

Key words: Logic programming, Datalog, MIP model, Production-Transportation, GAMS, TSP

1. Introduction

Many real world applications require to be formulated as mathematical programming problems. Such problems are usually tackled by first building a model, and then converting the model and the corresponding data into a low level instance. As a result, the instance can then be solved by a solver that returns a solution for further analysis.

An Algebraic Modeling Language (AML) is a language, sometimes part of a bigger system, for the development of such mathematical programming applications. It aims at expressing a mathematical programming problem in much the same way that a modeler does. It allows the user to describe mathematical models in a readable and symbolic form, resembling the well-known algebraic notation used to define sets, parameters, variables, objectives, and constraints.

Nowadays, a wide range of AMLs are available. Among the most widely used are AIMMS [8] (Advanced Integrated Multidimensional Modeling Software), GAMS [10, 25] (General Algebraic Modeling System), AMPL [28] (A Modeling Language for Mathematical Programming), FLOPC++ [31] (Formulation of Linear Optimization Problems in C++), ILOG Concert and OPL Technologies [39], and optimization platforms from MSF [43] (Microsoft Solver Foundation). These modeling languages are complex systems that require a certain learning curve in order to build large, complex models in a controlled fashion. Interested readers in similarities and contrasts among these AMLs are referred to [9].

A good modeling system makes virtually invisible to users most of the underlying complex data handling, instance generation, and computation. In the past, pure modeling systems with proprietary languages have dominated. Examples of these specialized modeling systems are illustrated in the top-left quadrant of the technology matrix in Fig. 1.

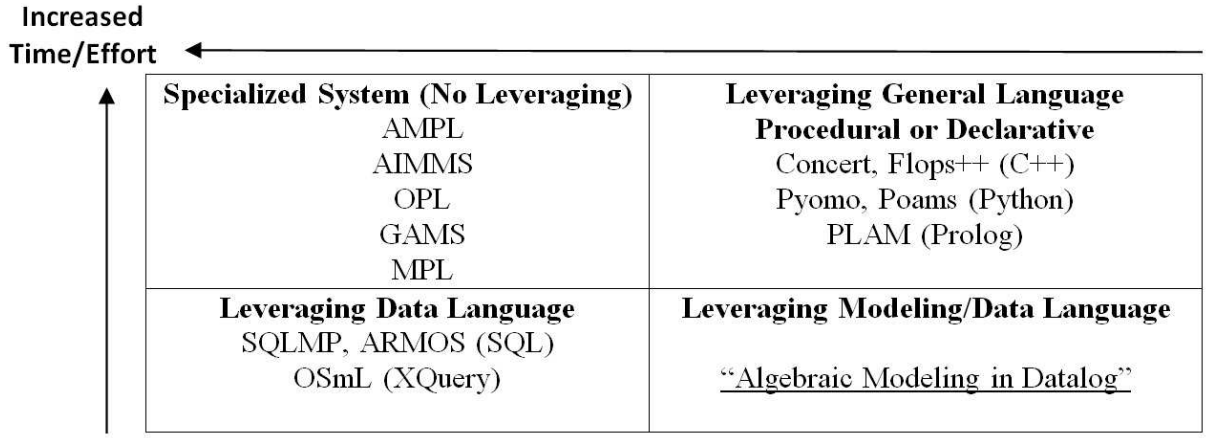

Figure 1 Algebraic Modeling Technology Matrix

One of the biggest challenges throughout the entire modeling and solution process is dealing with a large amount of data. One might even argue that mathematical modeling is mainly about data [31] and should therefore be driven by data. Indeed, a mathematical programming modeling language would not be of much practical use and could not gain acceptance unless it is closely integrated with corporate data, the sheer majority of which is stored within relational database systems or more modern key-value based database systems. While several specialized AML systems provide plug-ins for database access, such solutions do not easily provide flexibility in capturing the ever-changing business rules. This is specially pronounced for problems requiring changes to the modeling objects (e.g., add or remove nodes in a network) and introducing new rules on such objects. We provide a few such examples in Section 4.

We distinguish two disjoint research initiatives attempting to overcome these problems. The first group includes work based on the possibilities of adapting the common query language SQL to algebraic modeling (see, e.g., [5], [12], and [40]). However, while the data management part has been made easier and transparent, the extension of SQL to the algebraic modeling side always requires substantial development effort and the implementation is not easy to comprehend. Examples of the modeling systems that leverage existing data languages are listed in the bottom-left quadrant of the technology matrix in Fig. 1. This set of data languages include SQLMP [52], ARMOS (SQL) [5], and OSmL (XQuery) [47]. Analogously, the second group tries to adapt an existing general language that is close enough to an AML with fairly strong declarative and procedural capabilities, but comes short on the data side (see, e.g., [7], [31], [46, 47]). Examples of the modeling systems that leverage existing general languages are listed in the top-right quadrant of the technology matrix in Fig. 1. The list includes Concert Technologies [39], Flops++ [31], Pyomo [51], Poams [48], and PLAM [7]. A comprehensive list of AMLs and optimization tools can be found in [20].

As a conclusion, there is definitely a need for a solution with built-in language constructs that not only facilitates natural algebraic modeling but also provides integrated capabilities

with relational database handling. In this paper, we show that the basic functionalities and requirements of AMLs can be realized in any Datalog-like logic programming language.

Datalog is primarily a database query language. It is a rule-based logic programming language that based on the work present herein allows the application of the basic features of AMLs almost for free. Except for some syntactic sugar, no development effort is required to take advantage of Datalog-like language features and to meet the requirements from both the modeling and data management sides. As a result, we have a full programming language and a set of relational database tools that come with a Datalog system at our disposal, which can be used in all those cases where the built-in constructs of traditional modeling languages may not be sufficient. Our innovative use of Datalog fills the vacancy of the bottom right quadrant in the technology matrix in Fig. 1. A commercially available Datalog-based system has adopted and incorporated the ideas illustrated in this paper with hardly any extra development effort on the front end language and its supporting compiler.

Recent applications and extended languages based on Datalog can be found in [1, 4, 23, 30, 53], whereas Datalog-based industrial systems include ERBlox [6], MetaLogiQL [27], Google's Yedalog [11], Datomic [22], and the nascent EVE project 2 [24]. Interested readers in details of Datalog applications and extended languages are referred to [3, 19] and the references therein.

The main contributions of this work are twofold. We design a Datalog-based modeling language for solving mixed-integer linear optimization models. It essentially combines a highly effective Datalog-based database management system that allows to execute complex queries with simplistic logic clauses, and a set of intuitive and purely modeling tools to permit a straightforward inclusion of the advantages of AMLs. Note that our algebraic and syntactic modeling ideas are used in a commercially available platform, *LogicBlox*, also referred to as Datalog$^{\text{LB}}$ in this work.

Since the main characteristic shared among AMLs is their capability of providing a straightforward way to use explicit canonical forms that can be portable among different operating systems, as a second contribution of the present work, we conduct a comparison analysis between a commercially available algebraic modeling system, GAMS, and Datalog$^{\text{LB}}$'s embedded system. We opted for making comparisons and exemplifications with GAMS due to its maturity since it is considered the first and arguably the most commonly used AML within the engineering and science community. We also take advantage of GAMS regarding its syntactical similarities with other listed AMLs, which may help the reader define and understand the mathematical notations described in this work. Hence, we make use of two well-known optimization problems, namely a production-transportation and traveling salesman model, to explicitly exhibit drawbacks of AMLs when it comes to flexibility of business rules and data handling, and how Datalog$^{\text{LB}}$ circumvents this.

The organization of this paper is as follows. In Section 2, we give a short introduction to Datalog. In Section 3, we present the fundamentals of our implemented modeling interface, Datalog$^{\text{LB}}$, for solving mathematical programming problems. In Section 4, we provide a comparison analysis between Datalog$^{\text{LB}}$ and GAMS [25], in terms of both the modeling support and effective computation. Finally, concluding remarks and future work are given in Section 5.

2. Datalog

Datalog is a rule based Prolog-like query language for deductive database queries. Basically, a rule specifies what the computation should be rather than how to compute it. The specifics of

how the rule logic is executed correspond to the inner implementation of integrated computation engines.

Datalog is a subset of Prolog (originally invented in 1972 to provide a foundation for mathematics, see [14, 34]). It is based on the *first-order logic* targeted at providing a declarative language for expressing data structures, sophisticated calculations, analyses, integrity constraints, and transactional processing. Its major advantage stems from the fact that it preserves the SQL property of guaranteed termination, while expressing a significantly larger set of queries, including the recursive ones. This makes it of great interest and value in both the logic programming and database areas. Furthermore, the declarative nature of Datalog has a clear correspondence with mathematical logic, making it an apparent candidate as an AML.

Each rule in Datalog is a function-free Horn clause or predicate of the form

$$q(x) \leftarrow p_1(x_1), p_2(x_2), \cdots, p_n(x_n).$$

The term $q(x)$ is called the head (or conclusion) and it represents the relation containing the result of the rule query. The remaining terms relation containing the result of the rule query. The remaining terms $p_i(x_i), i = 1, \cdots, n$, are called sub-goals (or premises) and represent database relations. Functions $p_i(x_i)$ all evaluate to boolean values.

In Datalog programs, variables that appear in the head of the rule, must also appear in the body of the rule. In other words, variables in the head (conclusion) of the rule cannot be existentially quantified to prevent the system from having to make a choice for their value and then ensuring the consistency of that choice across the entire logical interpretation of the Datalog program. In terms of language expressiveness, this means that a number of algorithms (e.g., ones with non-polinomial running times) cannot be expressed. In terms of implementation, this means that no backtracking is necessary in the Datalog evaluation engine making the implementation potentially much more efficient.

This is a design choice: unlike a number of other logic programming languages, from the perspective of its original purpose, Datalog is primarily a database query language designed for the retrieval and management of data in relational database management systems.

In practice, any collection of Datalog rules that allows recursion can either be effectively rewritten to SQL and relational operators [2, 36, 37], or directly executed by a suitable Datalog evaluation engine. This is due to all modern database querying technologies using the concept of relational algebra, which in turn is based on the theory of the first-order logic. Main relational algebra operations include selection (σ), projection (Π), and Cartesian product (also called join, $\bowtie$). Since Datalog and its variants are implementations and extensions of the first-order logic, they naturally serve the querying purpose well.

A conditional selection operation $\sigma_{c(f_i)}(t[f_1, \cdots, f_i, \cdots, f_n])$ to select from table t all its n fields $f_1, \cdots, f_n$ with a condition c on field f_i can be expressed in Datalog as

$$selection_{(f_1,\cdots,f_i,f_n)} \leftarrow t(f_1, \cdots, f_i, \cdots, f_n), c(f_i).$$

A projection operation $\Pi_{(f_i)}(\sigma(t[f_1, \cdots, f_i, \cdots, f_n]))$ to select from table t particular field f_i can be expressed in Datalog as

$$selection_{projection(f_i)} \leftarrow t(f_1, \cdots, f_i, \cdots, f_n).$$

A join operation $\sigma_{f:f_i=g_j}(t_1[f_1, \cdots, f_i, \cdots, f_n]) \bowtie \sigma_{g:g_j=f_i}(t_2[g_1, \cdots, g_i, \cdots, g_m])$ to select all fields from table t_1 with n fields $f_1, \cdots, f_n$ and table t_2 with m fields $g_1, \cdots, g_m$ joined on f_i and g_j can be expressed in Datalog as

$$selection_{join(f_1,\cdots,f_i,\cdots,f_n,g_1,\cdots,g_i,\cdots,g_m)} \leftarrow t_1(f_1, \cdots, f_i, \cdots, f_n), t_2(g_1, \cdots, g_i, \cdots, g_m), f_i = g_j.$$

Note that by using these three operations, every transaction and application always terminates. Every predicate that follows can be derived from these three basic operations.

3. Mathematical Programming with Datalog

In this section, we describe some background and introductory concepts of our Datalog-based platform with support for mathematical programming. The Datalog modeling platform facilitates solving linear, mixed-integer programming problems by means of a set of syntactic constructs that allow specifying mathematical problems, and it provides an interface to several external optimization solvers.

A full mathematical programming model contains elements such as sets, indices, parameters, decision variables, objectives, constraints, and advanced algebraic expressions. A typical example is a linear program,

$$\min_x \{cx | Ax = b, x \geq 0\},$$

where x is a column vector of n variables, c is a row vector of n objective coefficients, A is an $m \times n$ matrix of constraint coefficients, and b is a column vector of m constraint right-hand side coefficients (with m constraints). The expression cx is the objective function and the equations $Ax = b$ are the constraints.

Similar syntax between AMLs and the mathematical notations of optimization problems facilitates a clear and concise representation of the problems. Given the declarative nature and various advantages of Datalog-like languages described above, we show how the essential requirements and basic functionalities of AMLs can be fulfilled naturally in Datalog. Familiar algebraic notation can be very easily incorporated by suitable Datalog predicates. The Datalog system then automatically generates the corresponding input for a mathematical programming solver.

3.1. Fundamentals of Datalog Modeling Platform

The ideas presented in this paper are implemented in a Datalog-based platform, LogicBlox [40]. The platform is designed to build enterprise-scale applications featuring analyses that require aggregation across very large data sets possibly combined with simulation, predictive analytics, and modeling techniques.

The LogicBlox platform is, first and foremost, a database management system in which a LogicBlox server manages one or more *workspaces*. A workspace is a database instance consisting of several components. Among the most common components in a workspace, we can find (a) *predicates*, which are analogous to tables containing data in relational database management systems, (b) *active programs*, which comprise collections of the logical rules contained in a workspace, and (c) constraints whose logical integrity is always maintained by the LogicBlox platform (analogous to database views in traditional relational databases).

The user's interaction with the workspace is managed through Datalog$^{\text{LB}}$'s modularity constructs, called *blocks*. A block is basically a piece of Datalog$^{\text{LB}}$ code which is submitted to the platform and integrated into the active program and schema of a workspace. Note that a label is given to each block so that they can be added to, or removed from the workspace. After type-checking and compilation, the rules and constraints defined over predicates contained in a block are incorporated into the *active program*.

With the use of blocks, the user can define tasks to be executed and compiled within Datalog$^{\text{LB}}$'s logic engine. Among the extensive set of possible tasks, the user can create new

predicates (data storage) and execute computations or retractions. Computations are a must to maintain the logical integrity of the database. Retractions are also fundamental tasks that allow the user to remove blocks from the workspace. Note that, when a block is removed, any rule for calculated predicates contained in it may in consequence be also removed from the active program along with any data derived from such rule. Blocks are considered inactive until the user incorporates them into the workspace at any given point in time, thus referring them to as *active blocks*.

A Datalog^{LB}program consists of a number of logical statements called *clauses*. A clause is composed of a *head* and a *body* (optional) followed by a period. Clauses may appear in the program in any order. The interaction between the LogicBlox workspace and the user proceeds over time through a series of commands (most of which are executed in *transactions*). Commands include operations such as:

- *installing a block*: adding a set of clauses in a Datalog^{LB} block to an active program permanently maintained by the database.
- *executing a block*: evaluating a set of clauses in a Datalog^{LB} block one time (without permanently incorporating its rules into the *active program*) for the purpose of updating the data stored in the workspace, or returning the results of the block as a query.
- *removing a block*: removing a set of clauses in a Datalog^{LB} block from the active program and deleting any data that may have been derived from it, and updating the state of the database to be logically consistent with the remaining active program.
- *querying* the contents of a predicate.

Most of these commands are executed transactionally: the user initiates a transaction, submitting one or more commands, and then commits. Each transaction provides the standard database ACID properties (Atomicity, Consistency, Isolation, Durability): all commands executed in a transaction are guaranteed an atomic, and consistent view of the database, isolated from any other commands initiated by other concurrent users. At the end of the transaction, the active program is evaluated and, if logically consistent, any chances made by it are durably stored in the database (transaction commits). Should the commands cause inconsistency (for example, by violating database constraints), the transaction is aborted and the workspace reverts to the (consistent) state before the transaction had been initiated.

3.1.1. Datalog^{LB} by Example All information in a LogicBlox workspace is organized into predicates, which denote key-value relations between finite *entities* (user-defined types) or built-in *data type*s (such as strings, integers, dates, and so on). An entity represents a user-defined finite set of abstract elements. In Datalog^{LB}, we declare entities by using the right-hand arrow (constraint) (`->`) operator.

From a purely logical perspective, the right-arrow in Datalog^{LB}context can be understood as a conditional or logical implication (if-then). From a more practical perspective, a right-arrow declaration can be used to declare the types of the arguments of a defined predicate. For example, we declare a new entity called 'NBAteam' by adding the following constraint to the workspace's active program:

```
NBAteam(x) ->.
```

This declaration asserts the existence of a unary predicate representing a set of objects `x` with the property 'being a NBAteam.' (In this case, e.g., the `NBAteam` predicate defines a finite set of National Basketball Association (NBA) teams.) Note that all statements in Datalog^{LB}must terminate with a period.

Data types in Datalog^{LB}can be divided into two categories, namely 1) user-defined types, and 2) built-in types. The former are specially declared unary entities that correspond to concrete objects (just like the `NBAteam` predicate defined above). The latter are the standard native types of modern programming languages: `int[n](x)`, `uint[n](x)`, `float[n](x)`, `string(x)`, and `datetime(x)`, where `n` represents a bit-width that can be 8, 16, 32, or 64.

A predicate may have a finite set of attributes of different types to account for the relationships between objects. One of a particular interest is *the reference mode predicate*, i.e., a predicate with a unique attribute that specifies a one-to-one relationship. In the NBA model, for example, we may need to uniquely identify each NBA team by its name, i.e., every instance of `NBAteam` must have a name, and no other `NBAteam` is allowed to have the same name. We can accomplished this by declaring a reference mode predicate in Datalog^{LB}as follows:

```
NBAteam(x), NBAteam:name[x]=s -> string(s), inv[NBAteam:name][s]=x.
```

where `NBAteam(x)` is a unary predicate, `NBAteam:name[x]` is a *functional predicate*, `string` is the reserved word encoding the standard string type, and `inv` is a meta-function to denote the inverse of a *functional predicate*, i.e., a binary predicate taking a variable as a function, which can be generalized as: `q[x] = y, inv[q][y]=x.` Note also that the key operators `(x)` and `[x]` used to define the predicates differ from each other in their functionality. For example, the former is used to declare special unary predicates that correspond to concrete objects (in this case for example, an NBAteam entity), whereas the latter is used in functional predicate to map entities (NBAteam) to specific-type-valued data (string names).

With the above statement, we indicate to Datalog^{LB}'s compiler that the predicate `NBAteam:name[x] = s` is a one-to-one mapping between objects `x` of `NBAteam` type and objects `s` of `string` type. More precisely, our one-to-one mapping imposes that each NBA team appearing in `NBAteam:name` has exactly one name, and that each name appearing in `NBAteam:name` has exactly one `NBAteam`

In addition to reference modes, we express an ordinary attribute of an entity by declaring a predicate that relates that entity to a primitive value, or to another entity type. For example, a 'coach' attribute on the `NBAteam` predicate is declared as follows:

```
NBAteam:Coach[x] = c -> NBAteam(x), string(c).
```

The logical implication of the right-arrow indicates that whenever a tuple (`x`, `c`) occurs in the predicate `NBAteam:Coach`, `x` is an `NBAteam` and `c` is a `string` type. In addition, the functional dependency from `x` to `c` is denoted by using the functional syntax `NBAteam:Coach[x] = c`, rather than the pure predicate notation `NBAteam:Coach(x, c)`. By doing so, we declare that each NBA team can have no more than one coach. The arguments in the angle braces `[x]` are called *keys*, whereas `c` is the value of the function. From an object-oriented perspective, the entity `NBAteam` and each of its elements may be thought of as a class and members of the class, respectively. From a relational database perspective, `NBAteam:Coach` represents a many-to-one relationship from `NBAteam(x)` to `string(c)`, i.e., every NBA team is associated with at most one string. (We assume that a coach can manage several teams.)

The case in which an entity has a set of values for one attribute represents a one-to-many relationship. For example, an NBA team might have a 'Players' attribute containing a set of NBA players. In Datalog^{LB}, we use a functional dependency in reverse—the entity is in the value argument, and the attribute is in the key argument—to represent this situation. We first declare a new entity called 'Players' as:

```
Players(p) ->.
```

Next, we declare the 'Players' attribute of the `NBAteam` as follows:

```
NBAteam:Players[p] = x -> NBAteam(x), Players(p).
```

This allows each NBA team `x` to have many players `p`, but each player still only belongs to one NBA team.

Entities may also have associations between them, thus representing a many-to-many relationship. For example, one player might be a friend of another player. Defining relationships between entities in Datalog^{LB}is straightforward. For example, the `friend` predicate can be declared as follows:

```
friend(x, y) -> Players(x), Players(y).
```

A predicate with an arity of 2 is also called a binary predicate. Note that ternary (3-place) or even more complex associations can be declared by simply specifying more arguments to the predicate.

Note that Datalog^{LB}also supports *negation* by means of the `!` operator. Examples and discussions on *negation* declarations are provided below.

Regarding data storage and handling in Datalog^{LB}, we describe a database consisting of defined series of entities and their corresponding relationships expressed as predicates. In a Datalog^{LB}database, each predicate is treated separately and populated with new elements or combinations of elements called facts.

Facts can be inserted, updated or removed to/from entities by means of simple constructs called *deltas*, which simply asserts the existence of an element with some particular properties. Deltas can be executed only on *extensional predicates* (EDB), i.e., predicates that are populated directly by user's transactions. (They have to be first declared by `->`.)

To support update rules, we enhance the definition of an EDB by making use of the `+`, `^` and `-` operators in Datalog^{LB}to indicate, respectively, the creation (insertion of a completely new key), update (change in the value of an existing key) and deletion (retraction of an existing key) of a fact. For example, *executing* the following delta against the workspace adds a new NBA team called 'Chicago Bulls':

```
+NBAteam(x), +NBAteam:name[x] = "Chicago Bulls".
```

Rules operating on *deltas* of predicates, rather than on entire predicates themselves can allso be installed and added to the active program. An example when this comes in handy is as follows. Assume we have the following two (EDB) predicates to define the salary and status of an NBA player:

```
Players:salary[p] = salary -> Players(p), float[64](salary).
Players:status[p] = status -> Players(p), string(status).
```

Then, the following rule fires whenever an NBA player's status is changed to 'laid-off' and sets the NBA player's salary to zero:

```
^Players:salary[p] = 0.0 <- ^Players:status[p] = "laid-off".
```

The rule states that whenever (in the current transaction) an NBA player's status has a new value of 'laid-off', then the NBA player's salary has a new value of zero. [1] It may be more intuitive to think of the `^Players:status` atom as the triggering event and the

[1] More pedantically, it states that whenever a change to the value of the `Player:status` predicate for a given person `p` occurs, then a change must be made to the value of `Players:salary` such that the `Players:salary` of `p` is equal to zero. The change to `Player:status` may occur through some other executed rules, by loading data through a user interface, or in some other way.

`^Players:salary` atom as the action. This causes Datalog^{LB}'s logic engine to evaluate all changes when the transaction commits, and only then are the changes accumulated into the corresponding predicates.

One of the most powerful features of Datalog^{LB}is the ability to define calculated predicates, also referred to as *intensional* predicates (IDB). In contrast to EDB, IDBs are derived by the rules already installed in the workspace and their data cannot be provided directly by the user, i.e., there are applications defining the IDBs' data in terms of the data in other predicates. IDB rules as usually added to the workspace's active program, as opposed to being executed as a command by EDBs. An example of a rule is shown in the following block:

```
person(x), person:name[x] = n->string(n), inv[person:name][n]=x.
parent(x,y) -> person(x), person(y).
male(x) -> person(x).
female(x) -> person(x).
father(x,y) <- parent(x,y), male(x).
mother(x,y) <- parent(x,y), female(x).
```

The first three lines declare, respectively, a set of people identified by names (the `person:name[x]` predicate), a parental relationship among people (the `parent(x,y)` predicate), and the specification that some people are males (the `male(x)` predicate). The last line is the rule, as indicated by the left-arrow, and it reads: whenever `x` and `y` belong to the `parent` relation, and `x` belongs to the set `male`, then `x` is the father of `y`. A rule consists of two *formulas*: the *head* (`father(x,y)`), and rule *body* `parent(x,y), male(x).` Atomic formulas (e.g., `father(x,y)`) may be combined with comma expressing *conjunction* in which both conjuncts must be true (`father(x,y)` *and* `male(x)`), or with *disjunction* (in which only one of the disjuncts must be true. For example, in the rule below, `x` is `y`'s grandmother, if `x` is the mother of some `z`, and `z` is either the mother of `y`, or the father of `y`:

```
grandmother(x,y) <- mother(x,z), (father(z,y) | mother(z,y)).
```

The left-arrow in the above rule represents a logical implication, i.e., whenever the right-hand-side (i.e., the body) of the rule is true, then the left-hand-side (called the head of the rule) is also true. The order of predicates in the head or body makes no difference to the results.

Note that the data for the calculated predicate is not directly editable by an application, thus guaranteeing that the rule remains true. In addition, any change made to the data in `person`, `parent`, or `male` etc. will result in an immediate (and usually incremental) recalculation of `father` according to the rule.

In addition to rules, Datalog^{LB}allows the users to state constraints. A constraint (usually written with implication from left to right (`->`) is a statement about what must be true in a given database. For example:

```
parent(x,y) -> !parent(y,x).
```

The above clause can be seen as a constraint imposed on the predicate `parent` stating that if `x` is the parent of `y`, then `y` cannot be parent of `x`. Note that the statement uses the `!` operator supported by Datalog^{LB}to represent *negation* (not) inside the constraint. Logically speaking, translations are equivalent to rules whose head is the logical falsity, and whose body is the negation of the meaning of the constraint. Applying some standard logical equivalences, we would rewrite the above as:

```
FALSE <- parent(x,y), parent(y,x)
```

(i.e., whenever x is a parent of y, and y is a parent of x, then *false*). Since the LogicBlox system must ensure that all clauses (rules) in the active program are consistent (i.e., that falsity can never be derived), the rule above states situations in which the active program cannot be evaluated to produce a consistent database. Practical implications of this are as follows: Suppose that `parent` is an EDB relation, whose data is supplied by the user, and not defined by rules, and the above constraint has been added to the active program. Then, any transaction that tries to insert data that violates the constraint above will be aborted by the LogicBlox platform, and unrolled to the (previously consistent) state before the start of the transaction.

Datalog$^{\text{LB}}$also supports disjunctions inside a constraint:

```
person(x), (male(x) | female(x)).
```

In the above statement the disjunction is represented by the vertical bar (`'|'`) as to indicate that a person is either a male or female.

As may be clear from the above, a subset of possible constraints in Datalog$^{\text{LB}}$are *type constraints*, such as: `parent(x,y) -> person(x), person(y).` Datalog$^{\text{LB}}$is strongly typed [54]: rules and constraints that would lead to type errors at runtime are statically rejected by the system at compile time. While some constraints (as the ones discussed immediately above) are checked at runtime, typing constraints are verified statically and incur no runtime cost.

In addition to regular Datalog rules, expressed as the first-order logic, Datalog$^{\text{LB}}$'s logic engine understands and evaluates *predicate to predicate mappings* (P2Ps), which specify relationships among predicates not easily expressed in first-order logic. This effectively overrides the semantics of the standard logical implication ($\leftarrow$) with a custom-defined implementation (provided by the implementers of the LogicBlox platform).

The most common P2Ps represent aggregations (`agg<< >>`). Consider the rule

```
NBAteam:Salary[x] = T <- agg<<T = total(s)>> Players:salary[p] = s,
                         NBAteam:Players[p] = x.
```

The P2P `agg` evaluates the variables on the right-hand side of the rule, for each value of `x` (indexed over `NBAteam`) and `p` (indexed over `Players`), then performs the operation of addition on all possible bindings of variable `s`, which concurrently binds variable `T` as the result of `total` (a built-in function in Datalog) and thus becoming the value of predicate `NBAteam:Salary`. Recall that operators `agg` and `total` are not part of standard Datalog, but their functionality can be implemented in "raw" Datalog.

The interaction between Datalog$^{\text{LB}}$'s logic engine and the workspace occurs during a transaction initiated by a Datalog query or other command. As depicted in Fig. 2, Datalog$^{\text{LB}}$'s logic engine updates the IDBs according to the installed rules in the blocks comprising the active program. In the figure, constraints are declared in the Datalog '*Source Program*', stored in '*Installed Blocks*' of P2P, and executed in '*Optimization Constraints*' when a transaction occurs. User query transactions are specified by means of the Datalog '*Source Program*' that has a designated answer predicate for querying the workspace state.

Next, we illustrate all of these features by means of a production-transportation model.

3.2. Algebraic modeling of a production-transportation model in Datalog$^{\text{LB}}$

The production-transportation model is to find the least cost production-shipping plan that meets demand requirements at markets and supplies at factories. There are costs given for

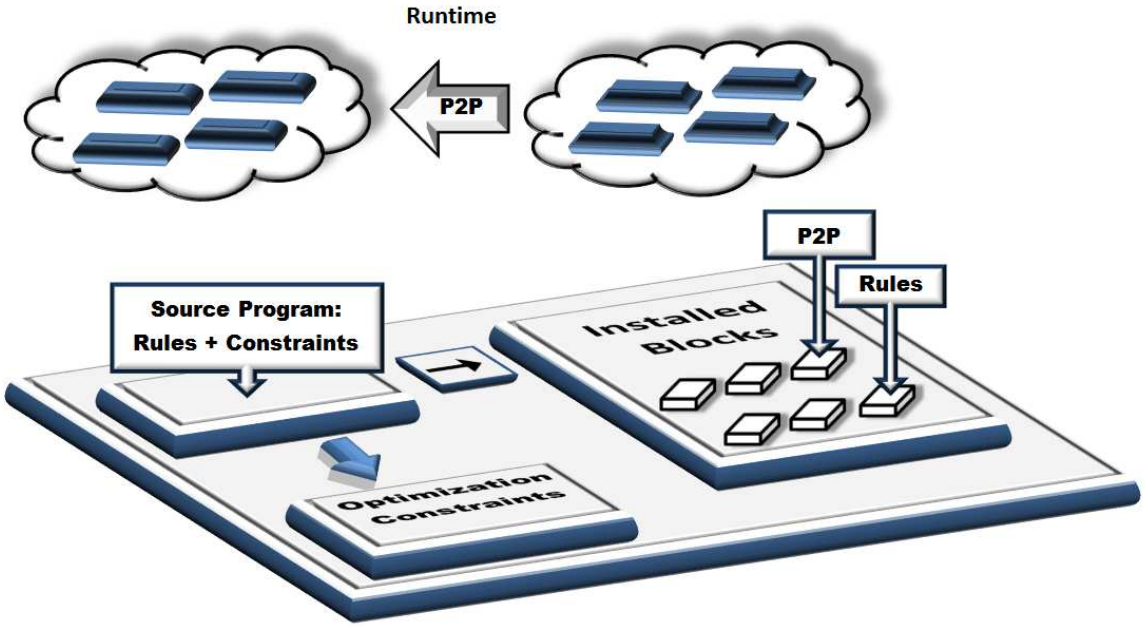

Figure 2 Datalog^{LB}'s logic engine for P2P mappings

producing and transporting. Details on production-transportation models can be found in, e.g., [10], [15], and [49].

The production-transportation model can be described as follows. Suppose that steel products are made at several mills, from which they are shipped to customers at various factories. For each mill, the decision variable captures the amount of each product to make. For each product on each arc with a mill as the origin and a factory as the destination, we need to obtain the amount of product to be shipped.

We start the algebraic model by defining the sets of products (`PROD`), origins (`ORIG`), and destinations (`DEST`). As parameters we are given `rate` as the tons produced per hour at the origins, `avail` as the hours available at origins, `demand` as amounts required at destinations, `makeCost` as manufacturing costs per ton of a product produced at an origin, and `transCost` as shipment costs per ton of a particular product shipped from an origin to a destination.

We declare two types of decision variables: `Make` as tons to be produced of each product at an origin, and `Trans` as tons to be shipped of each product from an origin to a destination.

The mathematical model can then be formulated as follows:

$$\min \sum_{\substack{i \in ORIG,\\ p \in PROD}} makeCost_{i,p} * Make_{i,p} + \sum_{\substack{i \in ORIG, j \in DEST,\\ p \in PROD}} transCost_{i,j,p} * Trans_{i,j,p} \quad (1)$$

s.t.:

$$\sum_{p \in PROD} \frac{1}{rate_{i,p}} * Make_{i,p} \leq avail_i, \qquad i \in ORIG, \quad (2)$$

$$\sum_{j \in DEST} Trans_{i,j,p} = Make_{i,p}, \qquad i \in ORIG, p \in PROD, \quad (3)$$

$$\sum_{i \in ORIG} Trans_{i,j,p} = demand_{j,p}, \qquad j \in DEST, p \in PROD. \quad (4)$$

Eq. (1) represents the objective function to be minimized as the sum of total costs of manufacturing and shipping, subject to time constraints (2) imposed by the number of hours

available at each origin, supply constraints (3) imposed by capacity restrictions, and demand constraints (4) imposed by the amount of each product required at each destination.

In AMLs, the data of a specific instance of a mathematical programming model consists of index sets and parameters. An index relates to a data item, which typically is either a coefficient or a variable. In the production-transportation model, the `PROD` set can consist of three types of items: `bands`, `coils`, and `plate`. In the Datalog context, we declare a unary relation `PROD` with members identified by names:

```
PROD(p), PROD:name(p:n) -> string(n).
```

The above statement uses the `PROD` predicate to define the concept of a "set" in algebraic modeling. Note that Datalog^{LB}characterizes a predicate symbol either as an identifier or a string. In `PROD:name(p:n)`, for example, we use the member field operator '`:`' both as an identifier and as a string. As an identifier, when enclosed in parentheses, '`:`' indicates that each member of the `PROD` set has a name of type `string`. As a string or reference mode (as it is called in object-relation modeling terminology), '`:`' is considered to be part of the ordinary characters that compose the contextual name of the predicate '`PROD:name`'.

We then specify the `PROD` data (facts), as follows:

```
+PROD(p), +PROD:name(p:"bands").
+PROD(p), +PROD:name(p:"coils").
+PROD(p), +PROD:name(p:"plate").
```

Similarly, we can declare the other two sets `ORIG` and `DEST` as follows:

```
ORIG(o), ORIG:name(o:n) -> string(n).
DEST(d), DEST:name(d:n) -> string(n).
```

Note that Datalog specifications are not unique and their syntax may vary based on the language implementation. In our model, the `avail` parameter is indexed over `ORIG` and we declare the `avail` unary predicate as

```
avail[o] = av -> ORIG(o), float[64](av).
```

to indicate that if `o` satisfies the `ORIG` predicate, it can be used to index `avail`, which evaluates to a 64 bit float typed value.

Moreover, since the `rate` parameter is indexed over more than one set (`ORIG` and `PROD`), we apply the conjunction rule (multiple set indexes) while separating by commas its declaration:

```
rate[o,p] = rv -> ORIG(o), PROD(p), float[64](rv).
```

Analogously, we declare the remaining parameters of the model as follows:

```
demand[d,p] = dm -> DEST(d), PROD(p), float[64](dm).
makeCost[o,p] = mc -> ORIG(o), PROD(p), float[64](mc).
transCost[o,d,p] = tc -> ORIG(o), DEST(d), PROD(p), float[64](tc).
```

Variables can be seen as special types of parameters with no syntactic differences. In the model we have two types of variables, one related to production and the other one related to transportation. We complete the declaration part of the model by establishing the decision variables in Datalog as

```
Make[o,p] = mk -> ORIG(o), PROD(p), float[64](mk), mk >= 0.
Trans[o,d,p] = tr -> ORIG(o), DEST(d), PROD(p), float[64](tr),tr>=0.
```

Reserved keywords are used to annotate that these predicates are variables:

```
lang:solver:variable(`Make).
lang:solver:variable(`Trans).
```

Regarding the syntactical specifications of the objective function and constraints, we emphasize that the relational aspect of logic programming is sufficient. Hence, we declare constraints (2) similarly as we did above with parameters:

```
sumTime[o] = st -> ORIG(o), float[64](st).
```

The constraints are all float-valued and indexed over the `ORIG` set. Type 'float' specifies the type of the left-hand side. In addition to this declaration statement (with the -> operator), we follow with a computation or rule deduction statement (with the <- operator):

```
sumTime[o] = st <- agg<<st=total(v)>> v=(1/rate[o,p])*Make[o,p].
sumTime[o] = t1, avail[o] = t2 -> t1 <= t2.
```

The first line above computes the left-hand side summation of the constraints. Keyword `agg` is a built-in aggregated method, which leverages on the built-in `total` function. They are used to model the summation in (2) and since the constraints are indexed over the `ORIG` set, the aggregated method operator will only sum over the `PROD` set, which is indexed by `p` in both the `rate` parameter and the `Make` variable.

The second line establishes the constraint relationship between the left- and right-hand side. The left-hand side is evaluated by both the `sumTime` predicate to `t1` and the `avail` predicate to `t2`, and the right-hand side indicates that `t1` cannot be larger than `t2`.

Similarly to (2), the supply constraints can be modeled as

```
sumSupply[o,p] = v -> ORIG(o), PROD(p), float[64](v).
sumSupply[o,p] = ss <- agg<<ss=total(m)>> m = Trans[o,_,p].
ORIG(o), PROD(p), sumSupply[o,p] = v1, Make[o,p]= v2 -> v1 = v2.
```

and the demand constraints can be modeled as follows:

```
sumDemand[d,p] = v -> DEST(d), PROD(p), float[64](v).
sumDemand[d,p] = sd <- agg<<sd=total(m)>> m = Trans[_,d,p].
DEST(d), PROD(p), sumDemand[d,p] = v1, demand[d,p]= v2 -> v1=v2.
```

Note that in Datalog^{LB}we can make use of the '_' operator to collect all occurrences of a predicate in rule heads into a single definition, see `sumSupply[o,p]` for example. This '_' operator comes in handy when we need to iterate over an unknown yet controlled index.

The objective function can be seen as a special constraint. Our objective is to minimize the total cost consisting of both production and transportation costs. The production cost is computed as

```
TotalMakeCost[] = tmc -> float[64](tmc).
TotalMakeCost[] = tmc <- agg<<tmc=total(v)>>
                          v = makeCost[o,p]*Make[o,p].
```

and the transportation cost is computed as

```
TotalTransCost[] = ttc -> float[64](ttc).
TotalTransCost[] = ttc <- agg<<ttc=total(v)>>
                         v = transCost[o,d,p]*Trans[o,d,p].
```

The total cost is thus computed as the sum of the two:

```
TotalCost[] = tc -> float[64](tc).
TotalCost[] = TotalMakeCost[] + TotalTransCost[].
```

Reserved keywords are used to indicate to Datalog^{LB}'s compiler that the predicate is minimized:

```
lang:solver:minimal('TotalCost).
```

The above clauses form the entire program block comprising the full model of the transportation-production problem in Datalog^{LB}. The entire implementation is provided in Appendix A.

3.3. Optimization in Datalog^{LB}

After the addition of a program into a dedicated workspace, Datalog^{LB}'s compiler transforms the problem specifications into the form readable by the selected solver. The selection of a solver in Datalog^{LB} is accomplished by using reserved keywords as follows:

```
optimization:solverName[] = "cbc".
```

where the `[]` notation is just syntactic sugar for the reserved keyword '`solverName`' with no arguments required, and 'cbc' refers to the open-source MIP solver COIN-OR Branch and Cut [13] (CBC) in Datalog^{LB}.

Solutions obtained by the solver can easily be accessed by means of simple queries. Note that by the inclusion of the mathematical model into the workspace, any transaction executed in Datalog^{LB}, including database updates and model extensions, automatically entails a call to the solver to update the current solution of the working instance. Note that model extensions correspond to the addition or adjustments of business rules of the existing optimization model currently stored in the dedicated workspace. These extensions can be introduced in form of, e.g., new constraints or variables to the Datalog source program.

Datalog^{LB}'s modeling platform incorporates the Optimization Services instance Language (OSiL) [41] to provide solvers with a standard way of accessing instance data and thus using various solvers as plug-ins. The OSiL standard is basically an XML schema that encodes instances of linear and mixed-integer programs. The Datalog^{LB} interface supplies the instance to the solver for the required computation, and then returns the results into Datalog predicates. The result recovering process is accomplished by means of the Optimization Services result Language (OSrL) that passes the results back to the application. This XML schema encodes solutions [41]. In addition to OSiL and OSrL standards, the complete framework also incorporates the Optimization Services option Language (OSoL) for communicating the algorithmic settings to solvers. This is also an XML schema. Fig. 3 shows the framework of the Datalog^{LB}'s modeling platform, including the interrelationships between the Datalog-based application programming interface and external processes and tools.

The mathematical programming capabilities of Datalog^{LB} are implemented as P2P, which relates the parameters of the optimization problem, EDBs and IDBs, to the variables of the optimization problem. In LogicBlox parlance, the parameters of the optimization problem comprise the body of the P2P and the variables comprise the head of the P2P.

By appearing in the head of the P2P, the variables of an optimization problem are automatically considered IDBs, even though they never appear in the head of a derivation rule in the original program. Thus, whenever any of the problem parameters change during a transaction—which may be caused directly by the addition of EDBs to the workspace, or by a rule deriving a change from related IDBs—the logic engine triggers the solver, which recomputes the values of the variables to solve the problem as it is specified by the P2P for the new values of the parameters. It is a restriction of Datalog^{LB}'s implementation that P2Ps cannot be involved in recursion, which enables the logic engine to trigger the solver only once per transaction for

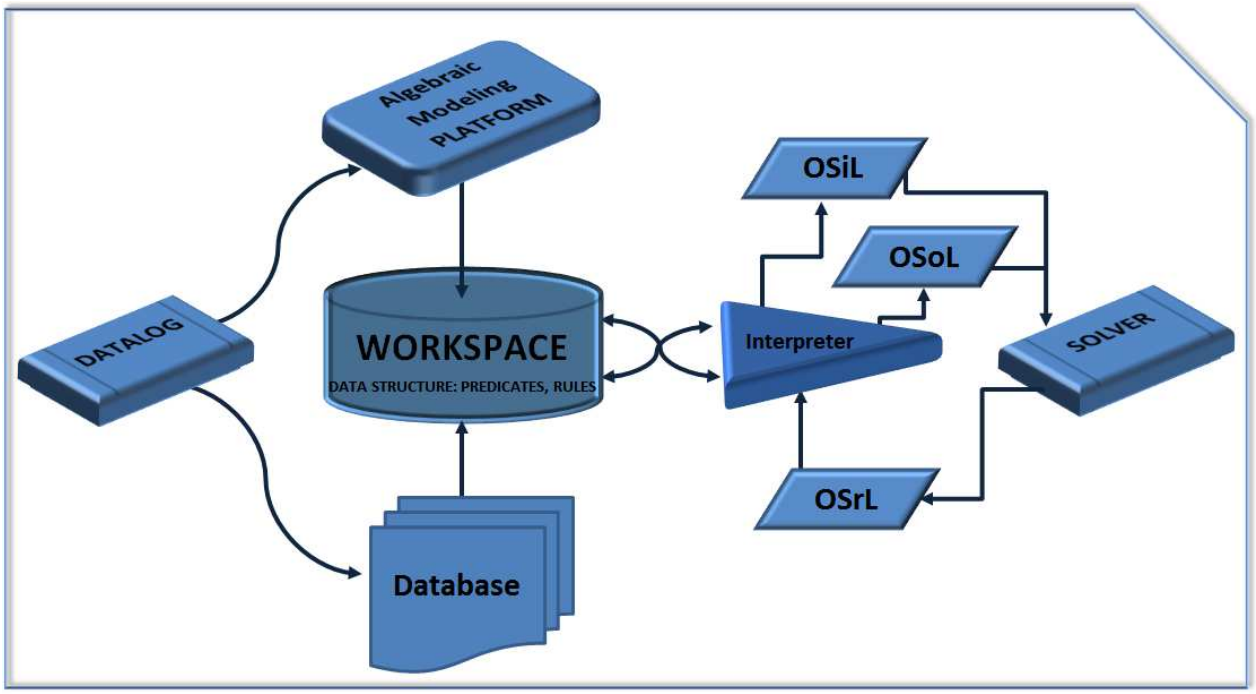

Figure 3 Framework of Datalog^{LB}'s modeling platform

a given problem. Relaxing this condition without losing Datalog^{LB}'s termination guarantee is one of the topics of future research.

To avoid repeated evaluations of optimization constraints—once as run-time constraints, and once as a part of the optimization problem specification—Datalog^{LB}'s compiler removes these constraints from the original program as one of the transformation steps. An important feature of this implementation is the ability to recognize subproblems of the entire mathematical program, which plays a key role on performance when the solver is called. This is accomplished by handling index sets separately from the model constraints, thus allowing Datalog^{LB}'s logic engine to evaluate the P2Ps incrementally, i.e, to only recompute the subproblems for which the parameters have changed.

4. Comparison Analysis: Datalog^{LB} vs GAMS

In this section, we present a comparison analysis between Datalog^{LB} and the General Algebraic Modeling System, GAMS [10, 25]. In particular, we analyze the capabilities of modeling and data handling on both platforms when solving an optimization problem, namely the *Traveling Salesman Problem* (TSP [17]).

4.1. The TSP Test Case

The TSP is one of the most intensively studied combinatorial optimization problems in computational mathematics specifically because it is so easy to describe and so difficult to solve. Given a set of n cities and distances between them, the TSP is to find the least cost tour visiting each city exactly once.

MIP formulations aimed to solve the TSP are, among others, due to Dantzig, Fulkerson and Johnson [16] (DFJ model) and Miller, Tucker and Zemlin [44] (MTZ model). The DFJ

model is an extension of a simple assignment model with the addition of subtours elimination constraints. The DFJ model requires a row generation procedure that basically generates one row after another with one design variable column only, thus providing in advance associated dual variables for each linking constraints (row).

Since the DFJ model requires 2^n subtour breaking constraints in the worst case, we implemented the MTZ model in Datalog$^{\text{LB}}$, which basically aims at breaking subtours with fewer constraints by making use of an extra variable u_i for each node i. By implementing the MTZ model, we also avoid the application of branch-and-bound techniques for possible fractional solutions to the DFJ model.

The mathematical model for the TSP, as well as its corresponding program in Datalog$^{\text{LB}}$ and GAMS are given in Appendix B.

In this work, we solve the TSP by means of the compact MIP formulation solely for modeling illustration purposes in Datalog$^{\text{LB}}$. Large-scale TSP's instances are solved by other means (see [29] for a complete historical development of the TSP).

We solved the test instance proposed by Dantzig, Fulkerson and Johnson [16] consisting of a set of 49 cities, one in each continental state of the USA and the District of Columbia. The list of cities is given in Appendix B.2. The corresponding triangular distance matrix among the cities is given in [16]. Note, however, that in [16], the triangular distance matrix for the 49-city problem is incomplete. We estimate our own distance values for cities A through G. The 49-city problem is solved by the open-source MIP solver COIN-OR Branch and Cut [13] (CBC) in Datalog$^{\text{LB}}$. Fig. 7 in Appendix B.2 shows the optimal tour with an objective of 694 units after roughly 24 CPU hours.

4.2. Assessments

Two sets of experiments were conducted. The first set consists of evaluating the computational effort into uploading data and building the mathematical model on both platforms for test instances of various sizes. The second set aims at assessing modeling efforts through select cases when tackling the TSP on both platforms. Such modeling endeavors refer to the ease to handle possible mathematical model extensions or data test case readjustments. This feature certainly represents one of the major challenges when selecting the proper modeling platform in solving real-life decision making problems. Typical examples concern adding and adjusting business rules. We show by an example that such efforts can be minimized with Datalog$^{\text{LB}}$.

All experiments were conducted on a 2.4 GHz Intel(R) Core(TM) i3 CPU processor with 4.0 GByte RAM under 64-bit MS-Windows operating system.

4.2.1. Data handling To achieve the goals of the first set of experiments, a set of TSP test instances were created by means of a Java-based generator. Values for the TSP triangular matrices correspond to positive uniformly distributed integer values over the range [1,10^5]. The test instances are given in Table 1, where the number of cities, the number of non-zero elements in the distance matrix file, and the storage file size are shown in Columns 1–3, respectively. Columns 4–5 show the CPU times spent on uploading data and creating the mathematical model for each test instance with Datalog$^{\text{LB}}$ and GAMS, respectively. The last column provides the real-physical disk space required for the dedicated workspace in Datalog$^{\text{LB}}$ for each test instance.

From the results shown in Table 1 we observe that GAMS' performance is underpar compared to Datalog$^{\text{LB}}$ when uploading the data and building the mathematical model for the

Table 1 Uploading data and creating the mathematical model for TSP instances

TSP test instance			Datalog^{LB}	GAMS	LB workspace$^{(**)}$
# of cities	Distance matrix$^{(*)}$	File size	CPU (sec)	CPU (sec)	storage
1,000	$1.50 \cdot 10^6$	6.5 MB	7	15	32.0 MB
5,000	$3.75 \cdot 10^7$	184.0 MB	231	725	700.0 MB
10,000	$1.50 \cdot 10^8$	747.0 MB	3,243	4,623	3,834.0 MB
15,000	$3.37 \cdot 10^8$	1,734.5 MB	21,886	35,653	6,207.5 MB
20,000	$6.00 \cdot 10^8$	3,122.5 MB	63,844	98,221	10,845.0 MB

$^{(*)}$ Number of non-zero elements in the distance matrix.
$^{(**)}$ 4.02 MB system predicates added to workspace in Datalog^{LB}.

very first time. However, in contrast to Datalog^{LB}, GAMS does not have a dedicated database storage. Instead, it requires that all data sets are either imported from external files by means of generators or scripts, or included manually in the file every time the mathematical model is to be created.

Assuming the simplest task of updating the value of only one entry of the TSP distance matrix, contrary to GAMS or any other AML with no dedicated workspace, Datalog^{LB} would simply update the value and automatically rerun the mathematical model without the need to upload the data again.

According to the CPU times shown in Table 1, working without a dedicated workspace may entail that a simple change represents a very time consuming process, whereas it may become a fruitless task when dealing with large-scale instances. In this regard, Datalog^{LB} easily outperforms AMLs with no dedicated workspace, such as GAMS, since an automatic update is executed exclusively on those predicates already stored in the dedicated workspace and which are involved with the current transaction. In other words, the incremental characteristic exposed by Datalog^{LB}'s logic engine basically allows the compiler to re-compute only those specific blocks (or predicates) involved with a change to the database due to a transaction (deletion, insertion or updated).

As a conclusion, while GAMS may require potentially the same amount of CPU time for any change made to the test instance, Datalog^{LB} updates both the database and model in such a way that no extra time is required for the task. Since a database may change constantly over time, a conspicuous advantage of using Datalog^{LB} over AMLs clearly arises when dealing with database intensive problems.

4.2.2. Modeling robustness and flexibility Our second set of experiments focused on modeling robustness and flexibility. We present three network modeling examples based on the TSP optimization model. We make use of the following predicate to formalize the examples:

```
Aij[i,j] = d -> city(i), city(j), int[64](d).
```

This predicate stores the original data of the triangular matrix for the TSP.

The first modeling task consists of removing a specific set of edges from the original TSP network. Consider a subnetwork composed of 5 nodes and 10 edges (as shown on the left side in Fig. 4 with the goal of removing two edges). Then, the modeling task can be easily accomplished within Datalog's context by simply adding the following predicate to the workspace:

```
NoLinks[i] = j -> city(i), city(j).
+NoLinks['A'] = 'E'.
+NoLinks['E'] = 'A'.
+NoLinks['B'] = 'D'.
```

```
+NoLinks['D'] = 'B'.
```

The `NoLinks` predicate stores the list of edges that are to be removed from the original network. Here we also see the use of an EDB rule prefixed by '+'.

Next, we update the network by executing only one transaction in Datalog^{LB}:

```
-Aij[i,j] = _ <- NoLinks[i] = j.
```

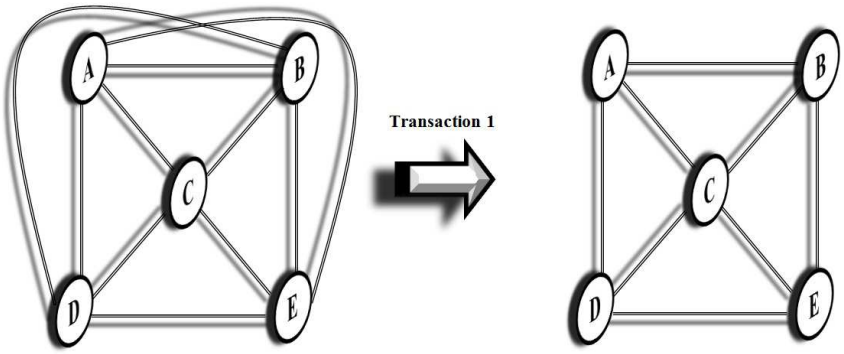

Figure 4 Example 1: Removing edges in Datalog^{LB}

In GAMS we may do this by manually fixing the decision variables defined for these arcs as follows. Let `i / A, B, C, D, E /`, `alias (i,j)` and `table c(i,j)` (included from an input file) be the subset of cities, the arc identifier and cost matrix in the GAMS model, respectively. Let $X(i,j)$ be the binary decision variable of arcs in the TSP network of the GAMS model. We then exclude the arcs by issuing the following:

```
x.fx('A','E') = 0;
x.fx('E','A') = 0;
x.fx('B','D') = 0;
x.fx('D','B') = 0;
```

For a relatively large test instance, fixing manually all arcs that are no longer part of the network may become an exhausting task. The appeal of the Datalog^{LB} approach is that the modification is done directly on the network and not at the modeling level by fixing the variables. Datalog^{LB} removes the variables through the concept of propagation of predicates.

In our second modeling case, we establish a specific set of nodes that are to be removed from the original network along with all of their incident edges. In Datalog^{LB} we do this by adding the following predicate to the workspace:

```
NoNodes(i) -> int[64](i).
+NoNodes('C').
```

The `NoNodes` predicate contains the nodes that are to be removed from the original network. We then add the following IDBs along with its rules to get all related edges corresponding to the `NoNodes` predicate (Fig. 5 illustrates this operation):

```
NoNodes:Edges[i,j] = d -> city(i), city(j), int[64](d).
NoNodes:Edges[i,j] = d <- Aij[i,j] = d, j>=i, NoNodes(i).
NoNodes:Edges[j,i] = d <- Aij[j,i] = d, j<i, NoNodes(i).
```

Analogously to our previous case, we update the network by executing the following transaction:

```
-Aij[i,j] = d <- NoNodes:Edges[i,j] = d.
```

Similarly to the previous case, we can update the TSP network in GAMS by manually managing the data in the GAMS model. However, unlike the previous example, fixing only the decision variables of arcs corresponding to the specific nodes that are to be removed from the original network is not enough. By doing so, GAMS would exclude the arcs but still keep the nodes as a part of the network model. The GAMS script might not require substantial effort for an experienced GAMS user, but it indeed requires a continual effort to maintain the network model.

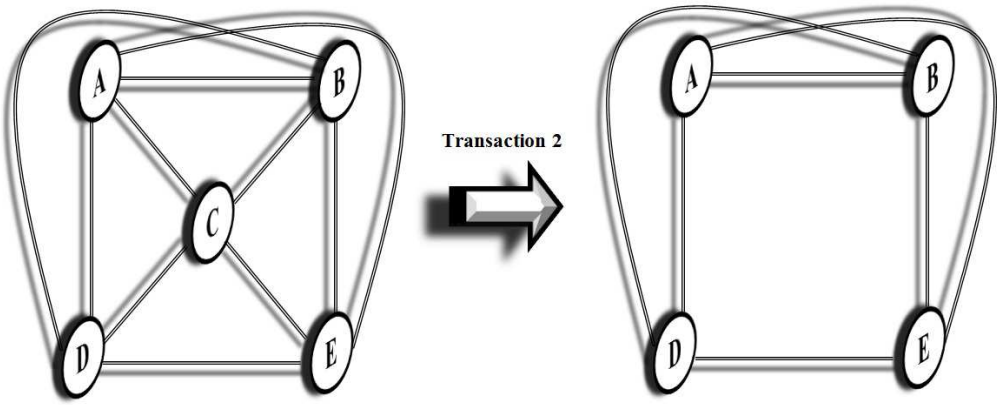

Figure 5 Example 2: Removing nodes and all related edges in Datalog^{LB}

In our final case we show how to handle a more complex network change with simple rules in Datalog^{LB}.

This time, given a base node, the modeling task consists of removing any node and all of the corresponding edges satisfying the following two conditions: (A) the node is within two edges from the base node, and (B) its degree, i.e., the number of edges incident to it, is less than or equal to δ.

Assume we have a subnetwork as shown on the left side in Fig. 6 with $\delta = 5$, and there is only one base node (node `O`) stored in the `BaseNode` predicate (shown as a black circle in Fig. 6):

```
+BaseNode('O').
```

In Datalog^{LB} we update this network in two steps. First, we find all paths starting at base node `O` that satisfy condition (A) and whose nodes (with exception of the base node) have degree less than or equal to five (condition (B)). In Datalog^{LB} we do this by adding the following *intensional predicates* to the workspace:

```
Degree[i] = v <- agg<<v=count()>> A[i,_]=_, city(i).
Find:Range2Paths[i,k,j] = dj <- A[i,k]=_, A[k,j]=_, Degree[j]=dj,
                        Degree[k]=dk, BaseNode(i), (dj<=5, dk<=5).
```

The `Degree` predicate evaluates the degree of each node (`city(i)`) in the network by applying the `count()` built-in function to the `A` predicate, which contains all arcs in the network. The `FindRange2Paths` predicate then uses the results stored in the `Degree` predicate to remove all paths accordingly.

The rule set for the `FindRange2Paths` predicate expresses that there exists a path in the form $\{(i,k),(k,j)\}$ with nodes k and j having degree $\leq \delta = 5$.

Next, we remove all but base node `O` and the corresponding related edges that are found on the paths. This is accomplished by the following predicates:

```
NodesToBeRemoved(n) <- (FindRange2Paths[_,n,_] = _;
                        FindRange2Paths[_,_,n] = _),!BaseNode(n).
EdgesToBeRemoved[i,j] = d <- Aij[i,j] = d, i=ii, j>=ii,
                             NodesToBeRemoved(ii).
EdgesToBeRemoved[j,i] = d <- Aij[j,i] = d, i=ii, j<ii,
                             NodesToBeRemoved(ii).
```

The network is then updated by executing the following transaction in Datalog^{LB}:

```
-Aij[i,j] = d <- EdgesToBeRemoved[i,j] = d.
-city(i) <- NodesToBeRemoved(i).
```

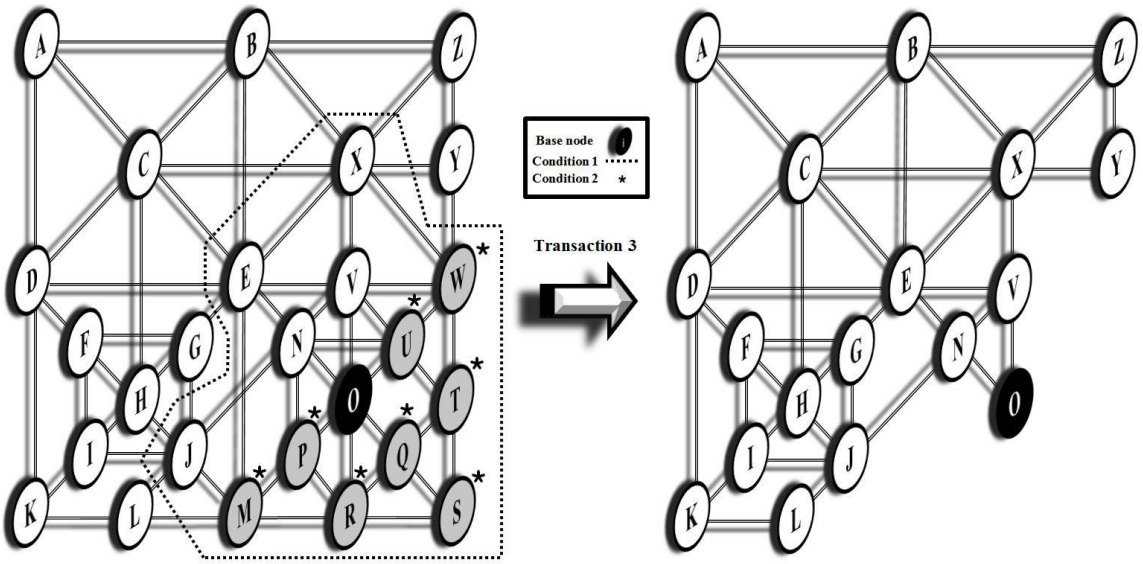

Figure 6 Example 3: A more complex example of network handling in Datalog^{LB}

For this particular network design problem, a GAMS user, on the other hand, is required to put a substantial effort to develop the necessary implementation using GAMS system's current tools. An implementation in GAMS is provided in Appendix C, and it requires using 'for' loops.

In all the previous examples, once the network related data (e.g., arcs, nodes and weights) are included in the GAMS model, any modification to the current network requires the following tasks: 1) either to make use of external tools to update the data accordingly or to make substantial changes to the GAMS input files whenever possible, and 2) construct again the mathematical model with the updated data.

Unlike GAMS, Datalog^{LB} has a highly effective modeling flexibility (as shown with the inclusion of specific rules to make the updates efficiently) that makes the tasks of data handling and model extensions very simple.

5. Conclusions and Future Work

In this paper, we presented the basic functionality needed for algebraic modeling of optimization problems built on top of a database system, yet special identifiers, operators and syntactic sugar are still in a developing phase to allow the current platform to mimic standard algebraic modeling systems with minor effort. Compared to existing algebraic modeling languages, the robustness of Datalog$^{\text{LB}}$ as a complete programming language with a pure declarative nature and native database support, offers great flexibility and extensibility in practical modeling.

The current implementation is limited to linear and mixed-integer programming, but it is intended to be enhanced with support for additional types of optimization problems and a wider range of solvers, from quadratic programming to meta-heuristics methods, all the way to Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers. (Details on these solvers can be found in, e.g., [26] and [32].)

References

[1] Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D. and Sears, R. (2011) Dedalus: Datalog in Time and Space. Book chapter, de Moor, O., Gottlob G., Furche, T. and Sellers, A.J. *(editors)*. Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK. pp. 262–281, Lecture Notes in Computer Science 6702, Springer, ISBN: ISBN 978-3-642-24205-2.

[2] Amornsinlaphachai, P., Rossiter, N. and Akhtar, M.A.(2006) Translating XML Update Language into SQL. Journal of Computing and Information Technology - CIT 14, Vol. 2, pp. 81–100. doi:10.2498/cit.2006.02.01

[3] Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pašalić,, E., Veldhuizen, T.L., and Washburn, G. (2015) Design and Implementation of the LogicBlox System. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15), Melbourne, Victoria, Australia, May 31-June 4.

[4] Arenas, M., Barceló, P., and Reutter, J. (2011) Datalog as Query Language for Data Exchange Systems. Book chapter, de Moor, O., Gottlob G., Furche, T. and Sellers, A.J. *(editors)*. Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK. pp. 302–320, Lecture Notes in Computer Science 6702, Springer, ISBN: ISBN 978-3-642-24205-2.

[5] Atamtürk, A., Johnson, E.L., Linderoth, J.T., and Savelsbergh, M.W.P. (2000) A Relational Modeling System for Linear and Integer Programming. Operations Research, Vol. 48(6), pp. 846–857.

[6] Bahmani, Z., Bertossi, L. and Vasiloglou, N. (2015) ERBlox: Combining Matching Dependencies with Machine Learning for Entity Resolution. In: Proceedings of the 9th International Conference on Scalable Uncertainty Management (SUM 2015), Quebec City, Canada, September 16-18.

[7] Barth, P. and Bockmayr, A. (1997) PLAM: ProLog and Algebraic Modeling. In: Proceedings of Practical Applications of Prolog, pp. 73–82.

[8] Bisschop, J. Paragon Decision Technology, AIMMS, http://www.aimms.com.

[9] Borraz-Sánchez, C. (2010) Optimization Methods for Pipeline Transportation of Natural Gas. Dissertation for the degree of PhilosophiæDoctor (PhD Thesis). Department of Informatics, University of Bergen, Norway.

[10] Brooke, A., Kendrick, D. and Meeraus, A. (1988) GAMS, A User's Guide. The Scientific Press, Redwood City, California.

[11] Chin, B., von Dincklage, D., Ercegovak, V., Hawkins, P., Miller, M.S., Och, F., Olston, C. and Pereira, F. (2015) Yedalog: Exploring knowledge at scale. In: Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL 2015), pp. 63–78, Dagstuhl, Germany.

[12] Choobineh, J. (1999) SQLMP: A Data Sublanguage for Representation and Formulation of Linear Mathematical Models. ORSA Journal of Computing, Vol. 3(4), pp. 358–375.

[13] COIN-OR Branch and Cut, CBC solver. (2005) An open-source mixed-integer program solver. IBM Research. https://projects.coin-or.org/Cbc
http://www.coin-or.org/Cbc/cbcuserguide.html

[14] Colmerauer, A. and Roussel, A. (1993) The birth of Prolog. ACM SIGPLAN Notices, Vol. 28 (3), pp. 37–52.

[15] Dantzig, G.B. (1963) In: Linear Programming and Extensions. Princeton University Press, Princeton, New Jersey.

[16] Dantzig, G.B., Fulkerson, D.R. and Johnson, S.M. (1954) Solution of a Large-Scale Traveling Salesman Problem. Operation Research, Vol. 2, pp. 393–410.

[17] Dantzig, G.B., Fulkerson, D.R. and Johnson, S.M. (1959) On a Linear-Programming, Combinatorial Approach to the Traveling-Salesman Problem. Operations Research, Vol. 7, pp. 58-66.

[18] DAMA, Data Management Association International. http://www.dama.org/i4a/pages/index.cfm?pageid=1

[19] Datalog 2010: Oxford, UK. Selected papers. http://dblp.uni-trier.de/db/conf/datalog/datalog2010.html

[20] Decision Tree For Optimization Software. http://plato.asu.edu/sub/tools.html

[21] Desaulniers, G., Desrosiers, J. and Solomon, M.M. (editors) (2005) Column Generation. Kluwer Scientific Publishers, pp. 163–196.

[22] Datomic, www.datomic.com

[23] Eisner, J. and Filardo, N.W. (2011) Dyna: Extending Datalog for Modern AI. Book chapter, de Moor, O., Gottlob G., Furche, T. and Sellers, A.J. *(editors)*. Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK. pp. 262–281, Lecture Notes in Computer Science 6702, Springer, ISBN: ISBN 978-3-642-24205-2.

[24] EVE, http://www.witheve.com

[25] GAMS Development Corporation. (2008) GAMS: The Solver Manuals. Washington, DC, USA.

[26] Ganesh, V. (2007) Decision Procedures for Bit-Vectors, Arrays and Integers. PhD. Thesis, Computer Science Department, Stanford University, Stanford, CA, USA.

[27] Green, T.J., Olteanu, D. and Washburn, G. (2015) Live programming in the LogicBlox system: A MetaLogiQL approach. In: Proceedings of the Very Large Data Bases (VLDB Endowment), Vol. 8 (12), Kohala Coast, Hawaii.

[28] Fourer, R., Gay, D.M. and Kernighan, B.W. (1993) AMPL. A Modeling Language for Mathematical Programming. The Scientific Press, Redwood City, CA.

[29] Hoffman, A.J. and Wolfe, P. (1985) "History" in the Traveling Salesman Problem. Lawler, Lenstra, Rinooy Kan and Shmoys, eds., Wiley, 1-16.

[30] Huang, S.S., Green, T.J. and Loo, B.T. (2011) Datalog and emerging applications: An interactive tutorial. In SIGMOD, pp. 1213–1216, 2011.

[31] Hultberg, T. FLOPC++, An Algebraic Modeling Language Embedded in C++, https://projects.coin-or.org/FlopC++.

[32] Jha, S., Limaye, R. and Seshia, S.A. (2009) Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. Lecture Notes in Computer Science, Vol. 5643, pp. 668–674.

[33] Karp, R.M. (1972) Reducibility among Combinatorial Problems. Plenum Press, pp. 85-103.

[34] Kowalski, R.A. (1988) The early years of logic programming. Communications of the ACM, Vol. 31 (1), pp. 38–43.

[35] Kristjansson, B. Maximal Software, MPL Modeling System, http://www.maximal-usa.com.

[36] Krishnamurthy, R. (2004) XML-to-SQL Query Translation. Dissertation for the degree of PhilosophiæDoctor (PhD Thesis). University of Wisconsin - Madison, USA.

[37] Krishnamurthy, R., Chakaravarthy, V.T., Kaushik, R. and Naughton, J.F. (2004) Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In: Proceedings of the 20th International Conference on Data Engineering, pp. 42–53, March. doi:10.1109/ICDE.2004.1319983

[38] Kristjansson, B. Maximal Software, MPL Modeling System, http://www.maximal-usa.com.

[39] ILOG, OPL, http://www.ilog.com/products.

[40] LogicBlox, http://www.logicblox.com.

[41] Ma, J. (2005) Optimization services (OS), a general framework for optimization modeling systems. Ph. D. Dissertation, Department of Industrial Engineering & Management Sciences, Northwestern University, Evanston, IL, USA.

[42] Ma, J. and Martin, K. (2004) OSmL, Optimization Services Modeling Language, https://www.coin-or.org/OS/projects_osml.html.

[43] Microsoft Solver Foundation. Amsterdam Optimization and Microsoft Solver Foundation.
http://code.msdn.microsoft.com/solverfoundation
http://www.amsterdamoptimization.com/solverfoundation.html

[44] Miller, C.E., Tucker, A.W. and Zemlin, R.A. (1960) Integer Programming Formulations of Traveling Salesman Problems. Journal of ACM, Vol. 7 (4), pp. 326-329.

[45] Mitra, G., Lucas, C., Moody, S. and Kristjansson, B. (1995) Sets and Indices in Linear Programming Modelling and Their Integration with Relational Data Models. Journal of Computational Optimization and Application, Vol. 4(3), pp. 263–292.

[46] Optimization Services (OS), http://www.optimizationservices.org.

[47] Optimization Services modeling Language (OSmL) http://www.coin-or.org/OS/projects_osml.html

[48] POAMS, Python Object-Algebraic Modeling System. http://www.coin-or.org/OS/projects_overview.html

[49] Pochet, Y. and Wolsey, L.A. (2006) Production Planning by Mixed Integer Programming. Springer Series in Operations Research and Financial Engineering. Springer-Verlag New York, Inc.

[50] PULP, http://130.216.209.237/engsci392/pulp/FrontPage.

[51] Pyomo, Flexible modeling of optimization problems in Python. http://www.pyomo.org

[52] SQLMP (1995) Introduction to NonStop SQL/MP. Hewlett-Packard Development Company L.P., Tandem Computers Incorporated Data Management Library. http://h10032.www1.hp.com/ctg/Manual/c02148175

[53] Smaragdakis, Y. and Bravenboer, M. (2011) Using Datalog for fast and easy program analysis. Book chapter, de Moor, O., Gottlob G., Furche, T. and Sellers, A.J. *(editors)*. Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK. pp. 245–251, Lecture Notes in Computer Science 6702, Springer, ISBN: ISBN 978-3-642-24205-2.

[54] Zook, D., Sarna-Starosta B., and Pasalic, E. (2009) Typed Datalog. Practical Aspects of Declarative Languages: 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009. Proceedings. Springer Berlin Heidelberg.

Appendix

A. The Transportation-Production Model

The following block declares the index sets and parameters for the transportation-production model in Datalog^{LB} as described in Section 3.2. Note that the block is contained between `<doc> .. </doc>` with a block name (`TransProdModelDeclaration`) appended to it. The block's name can basically be used by the user to active it or remove it from the dedicated workspace by executing specific transactions. In the examples below, the transaction is initiated by the user with the keyword `transaction`, and then, the block is officially activated once the transaction is committed by using the keyword `commit`.

```
transaction
addBlock --blockName TransProdModelDeclaration <doc>

// INDEX SETS:
ORIG(p), ORIG:name(p:n) -> string(n).  // Set of origins
PROD(p), PROD:name(p:n) -> string(n).  // Set of products
DEST(p), DEST:name(p:n) -> string(n).  // Set of destinations

// PARAMETERS:
// avail = hours available at origins (indexed over ORIG)
// We declare the avail unary predicate as:
avail[o] = av -> ORIG(o), int[64](av).

// rate = tons produced per hour at the origins (indexed over ORIG and PROD)
// We declare the rate predicate as:
rate[o,p] = rv -> ORIG(o), PROD(p), float[64](rv).

// demand = amount of each product required at destinations (indexed over DEST and PROD)
// We declare the demand unary predicate as:
demand[d, p] = dp -> DEST(d), PROD(p), float[64](dp).

// makeCost = manufacturing costs per ton of a product produced at an origin
// (indexed over ORIG and PROD)
// We declare the makeCost predicate as:
makeCost[o,p] = mC -> ORIG(o), PROD(p), float[64](mC).

// transCost = shipment costs per ton of a particular product shipped from an origin to
// a destination (indexed over ORIG, DEST and PROD)
// We declare the transCost predicate as:
transCost[o,d,p] = sC -> ORIG(o), DEST(d), PROD(p), float[64](sC).

</doc>
commit
```

The following active block adds the transportation-production model to the workspace in Datalog^{LB}.

```
transaction
```

```
addBlock --blockName TransProdModel <doc>

// VARIABLES DECLARATION:
// (Note: Variables are defined using predicates, which are parametrized by index entities
//        with integer reference modes.)

//We declare two types of decision variables:
//(1) Make = tons to be produced of each product at an origin (indexed over ORIG and PROD)
Make[o,p]= mk -> ORIG(o), PROD(p), float[64](mk), mk>=0.
ORIG(o), PROD(p) -> Make[o,p] = _.
lang:solver:variable(`Make).

//(2) Trans = tons to be shipped of each product from an origin to a destination
// (indexed over ORIG and DEST)
Trans[o,d,p] = tr -> ORIG(o), DEST(d), PROD(p), float[64](tr), tr >=0.
ORIG(o), DEST(d), PROD(p) -> Trans[o,d,p] = _.
lang:solver:variable(`Trans).

// CONSTRAINTS DEFINITION:
// <<Time constraints>> imposed by the number of hours available at each origin
//                      (indexed over ORIG) while making use of avail[o]
sumTime[o] = st -> ORIG(o), float[64](st).

// (Note: After the syntactical specification of the constraint, we follow with the computation
//        or rule deduction statement with the '<-' operator.)
sumTime[o] = st <- agg<<st=total(v)>> v=(1/rate[o,p])*Make[o,p].
sumTime[o] = t1, avail[o] = t2 -> t1 <= t2.

// (Note: The first line above computes the left-hand side summation of the time constraints.
//        The 'agg' is a built-in aggregated method, which leverages on the built-in 'total'
//        function. They are used to model the summation in the constraint. Since the constraint
//        is indexed over the ORIG set, the aggregated method operator will only sum over the
//        PROD set, which is indexed by p in both the rate parameter and the Make variable.)

// (Note: The second line establishes the constraint relationship between the lhs and the rhs.
//        The lhs is evaluated by both the 'sumTime' predicate to t1 and the 'avail' predicate
//         to t2, and the rhs indicates that t1 cannot be larger than t2.)

// <<Supply constraints>> imposed by capacity restrictions
sumSupply[o,p] = v -> ORIG(o), PROD(p), float[64](v).
sumSupply[o,p] = ss <- agg<<ss=total(m)>> m = Trans[o,_,p].
ORIG(o), PROD(p), sumSupply[o,p] = v1, Make[o,p]= v2 -> v1 = v2.

// <<Demand constraints>> imposed by the amount of each product required at each destination
sumDemand[d,p] = v -> DEST(d), PROD(p), float[64](v).
sumDemand[d,p] = sd <- agg<<sd=total(m)>> m = Trans[_,d,p].
DEST(d), PROD(p), sumDemand[d,p] = v1, demand[d,p]= v2 -> v1 = v2.

// OBJECTIVE FUNCTION:
// (Note: It can be seen as a special constraint with specific identifiers --external
//        configuration parameters-- to indicate whether the objective in the model is
//        minimization or maximization. Our objective is to minimize the total cost
//        consisting of production and transportation costs.)

//The production cost is computed as
TotalMakeCost[] = tmc -> float[64](tmc).
TotalMakeCost[] = tmc <- agg <<tmc=total(v)>> v=makeCost[o,p]* Make[o,p].

//and the transportation cost is computed as
TotalTransCost[] = ttc -> float[64](ttc).
TotalTransCost[] = ttc <-agg<<ttc=total(v)>> v=transCost[o,d,p]*Trans[o,d,p].

//The total cost is thus computed as the sum of the two:
TotalCost[] = tc -> float[64](tc).
TotalCost[] = TotalMakeCost[] + TotalTransCost[].
lang:solver:minimal(`TotalCost).   // Defining the sense of the objective function
optimization:solverName[]="cbc".   // Setting the solver's name

</doc>
commit
```

The following block executes the data loading procedures of sets and parameters for solving the transportation-production model in Datalog$^{\text{LB}}$.

```
// Importing data model for TransProdModel in Datalog LB
transaction
exec <doc>

// Read in product info (columns 1-5): id, name, orig, dest, and avail
// declare indexes and load csv file
_inTransProd(id, prod, orig, dest, avail) -> int[64](id), string(name),
string(orig), string(dest), int[64](av).
lang:physical:storageModel[`_inTransProd] = "DelimitedFile".
lang:physical:filePath[`_inTransProd] = "TransProd_sets.csv".
lang:physical:hasColumnNames[`_inTransProd] = false.

// define corresponding deltas to load
+PROD(_) {
+PROD:name[]=name,
+ORIG:name[]=orig,
+DEST:name[]=dest,
+avail[]=av } <- _inTransProd(_, name, orig, dest, av).

// Read in TransProd (TP)'s parameters from a csv file
// declare indexes and load csv file
_inTPparams(idxO, idxD, idxP, rv, dp, mC) -> int[64](idxO), int[64](idxD), int[64](idxP),
float[64](rv), float[64](dp), float[64](mC).
lang:physical:storageModel[`_inTPparams] = "DelimitedFile".
lang:physical:filePath[`_inTPparams] = "TransProd_params.csv".
lang:physical:hasColumnNames[`_inTPparams] = false.

// define corresponding deltas to load
+rate[o,p] = rv, +demand[d,p] = dp, +makeCost[o,p] = mC
<- _inTPparams(idxO, idxD, idxP, rv, dp, mC).

/ Read in TransProd (TP)'s cost matrix from a csv file
// declare indexes and load csv file
_inTPmatrix(idxO, idxD, idxP, sC) -> int[64](idxO), int[64](idxD), int[64](idxP), float[64](sC).
lang:physical:storageModel[`_inTPmatrix] = "DelimitedFile".
lang:physical:filePath[`_inTPmatrix] = "TransProd_CostMatrix.csv".
lang:physical:hasColumnNames[`_inTPmatrix] = false.

// define corresponding deltas to load
+transCost[o,d,p] = sC <- _inTPmatrix(idxO, idxD, idxP, sC).

</doc>
commit
```

B. The Traveling Salesman Problem

Given an arc-weighted complete graph $G = (V, A, w)$, where V is the set of nodes (cities), A is the set of arcs linking a pair of cities $u, v \in V$, and $w : A \longmapsto \mathbb{R}^+$ is the cost function assigning each $(u, v) \in A$ weight w_{uv}, the problem is to find a minimum weight tour in G. The following variables are required for the MTZ model:

$$x_{ij} = \begin{cases} 1 & \text{1 if arc from node } i \text{ to node } j \text{ is selected,} \\ 0 & \text{otherwise,} \end{cases}$$

$$u_i = \text{the position of node } i \text{ in the tour.}$$

The MTZ formulation reads:

$$\min \sum_i \sum_j c_{ij} x_{ij} \tag{5}$$

$$\text{s.t.:} \quad \sum_i x_{ij} = 1, \quad j, \tag{6}$$

$$\sum_j x_{ij} = 1, \quad i, \tag{7}$$

$$u_i + 1 \leq u_j + n(1 - x_{ij}), \; i = 2, \cdots, n, i \neq j, j = 2, \cdots, n, \tag{8}$$

$$u_i \leq n - 1 - (n-2)x_{1i}, \; i \geq 2, \tag{9}$$

$$u_i \geq 1 + (n-2)x_{i1}, \quad i \geq 2, \tag{10}$$

$$x \in \{0, 1\}, u \geq 0. \tag{11}$$

The formulation assumes that the tour starts at node 1. Eq. (5) is the objective function of the TSP model aimed at minimizing the total cost of the tour. Constraints (6) and (7) are the assignment constraints, for entering and leaving each node exactly once, respectively. Constraints (8)–(10) are the subtour elimination constraints.

B.1. The MTZ Model for TSP in Datalog$^{\text{LB}}$

The following block declares the index sets and distance (cost) table for the symmetric TSP model in Datalog$^{\text{LB}}$. Note that the block is enclosed between `<doc> .. </doc>` with a block name (`TSPModelDeclaration`) appended to it. The user can then use the block's name to active it or remove it from the dedicated workspace. In the example below, this is accomplished by initiating a transaction with the keyword `transaction`, and then, activating the block with the keyword `commit`, which basically indicates to Datalog$^{\text{LB}}$'s logic engine that the transaction is committed.

```
transaction
addBlock --blockName TSPModelDeclaration <doc>

//Total number of cities, 'intensional' predicate:
Nc[] = n -> int[64](n).
Nc[] = n <- agg<<n=count()>> city(_).

//Set of cities:
city(c)->.
city:id[c] = n -> city(c), int[64](n).
city:name[c] = n -> city(c), string(n).

//Arcs
```

```
Aij[i,j] = d -> int[64](i), int[64](j), int[64](d).
A[i,j] = d -> city(i), city(j), int[64](d).
A[i,j] = d <- city:id[i] = ii, city:id[j] = jj, Aij[ii,jj] = d.

</doc>
commit
```

The following block adds the MTZ model to the workspace for solving the TSP in Datalog^{LB}.

```
transaction
addBlock --blockName MTZModelforTSP <doc>

// VARIABLE DECLARATION:
// (Note: Variables are defined using predicates, which are
//        parametrized by index entities with integer refmods.)

//Decision variables:
//(1) Xij = 1 if arc from node i to node j is selected, Xij=0 otherwise.
Xij[i,j] = x -> city(i), city(j), int[64](x), x<=1, x>=0.
city(i), city(j) -> Xij[i,j] = _.
lang:solver:variable(`Xij).

//(2) Ui = the position of node i in the tour
Ui[i] = v -> city(i), int[64](v), v>=0.
city(i) -> Ui[i] = _.
lang:solver:variable(`Ui).

// CONSTRAINTS:
// Eq.(6): Enter each node once: sum_i Xij = 1
sumLHS[j] = v -> city(j), int[64](v).
sumLHS[j] = v <- agg<<v=total(r)>> r=Xij[i,j], city(i), i!=j.
city(j), sumLHS[j] = v -> v = 1.

// Eq.(7): Leave each node once: sum_j Xij = 1
sumLHS2[i] = v -> city(i), int[64](v).
sumLHS2[i] = v <- agg<<v=total(r)>> r=Xij[i,j], city(j), i!=j.
city(i), sumLHS2[i] = v -> v = 1.

// Subtour elimination constraints:
// Eq.(8):  u_i + 1 <= u_j + n(1-Xij), for all i=2,...,n, i!=j, j=2,...,n,
city(i), city(j), v1 = Ui[i] - Ui[j] + Nc[]*Xij[i,j], v2= Nc[]-1, i!=j, i>=2, j>=2 -> v1 <= v2.
// Eq.(9):  u_i <= n - 1 - (n-2)*X1i, for all i >= 2
city(i), v1 = Ui[i], v2 = Nc[] - 1 - (Nc[]-2)*Xij[1,i], i>=2 -> v1 <= v2.
// Eq.(10):  u_i >=1 + (n-2)*Xi1, for all i >= 2
city(i), v1 = Ui[i], v2 = 1 + (Nc[]-2)*Xij[i,1], i>=2 -> v1 >= v2.

// OBJECTIVE FUNCTION:
// TSP model aims at minimizing total cost of the tour

TourLength[] = tc -> int[64](tc).
TourLength[] = tc <- agg<<tc=total(v)>> v=A[i,j]*Xij[i,j], i!=j.
lang:solver:minimal(`TourLength).
optimization:solverName[]="cbc".

</doc>
commit
```

The following block executes the data loading procedures for solving the symmetric TSP model in Datalog^{LB}: loading a 10×10 TSP instance.

```
// Importing data model for TSP in Datalog LB: TSPReader10x10.lb
transaction
```

```
exec <doc>

// Read in cities' id (column 1) and name (column 2) from a csv file
// declare indexes and load csv file
_inCities(id, name) -> int[64](id), string(name).
lang:physical:storageModel[`_inCities] = "DelimitedFile".
lang:physical:filePath[`_inCities] = "TSP_10x10cities.csv".
lang:physical:hasColumnNames[`_inCities] = false.

// define corresponding deltas to load
+city(_) {
+city:id[] = id,
+city:name[] = name } <- _inCities(id, name).

// Read in distance matrix from a csv file
// declare indexes and load csv file
_inD(indexi, indexj, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10) -> int[64](indexi), int[64](indexj),
int[64](a1), int[64](a2), int[64](a3), int[64](a4), int[64](a5), int[64](a6), int[64](a7),
int[64](a8), int[64](a9), int[64](a10).
lang:physical:storageModel[`_inD] = "DelimitedFile".
lang:physical:filePath[`_inD] = "TSP_10x10distanceMatrix.csv".
lang:physical:hasColumnNames[`_inD] = false.

// define corresponding deltas to load
+Aij[i,0] = a0, +Aij[i,1] = a1, +Aij[i,2] = a2, +Aij[i,3] = a3, +Aij[i,4] = a4,
+Aij[i,5] = a5, +Aij[i,6] = a6, +Aij[i,7] = a7, +Aij[i,8] = a8, +Aij[i,9] = a9
<- _inD(i, _, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9).

</doc>
commit
```

B.2. A TSP test instance in Datalog^{LB}

We solve the test instance proposed by Dantzig, Fulkerson and Johnson [16]. Table 2 shows the list of the cities.

Table 2 List of names of selected cities, a 49-city TSP instance

1. Manchester, NH	14. Seattle, WA	27. Topeka, KS	40. Washington, DC
2. Montpelier, VT	15. Portland, OR	28. Oklahoma City, OK	41. Boston, MA
3. Detroit, MI	16. Boise, ID	29. Dallas, TX	42. Portland, ME
4. Cleveland, OH	17. Salt Lake City, UT	30. Little Rock, AR	A. Baltimore, MD
5. Charleston, WV	18. Carson City, NV	31. Memphis, TN	B. Wilmington, DE
6. Louisville, KY	19. Los Angeles, CA	32. Jackson, MS	C. Philadelphia, PA
7. Indianapolis, IN	20. Phoenix, AZ	33. New Orleans, LA	D. Newark, NJ
8. Chicago, IL	21. Santa Fe, NM	34. Birmingham, AL	E. New York, NY
9. Milwaukee, WI	22. Denver, CO	35. Atlanta, GA	F. Hartford, CT
10. Minneapolis, Mn	23. Cheyenne, WY	36. Jacksonville, FL	G. Providence, RI
11. Pierre, SD	24. Omaha, NE	37. Columbia, SC	
12. Bismarck, ND	25. Des Moines, IA	38. Raleigh, NC	
13. Helena, MT	26. Kansas City, MO	39. Richmond, VA	

The MTZ model for this TSP test instance comprises 1,723 rows, 1,762 variables and 8,284 non-zero elements in the constraint matrix. The model takes the form of an integer, linear programming (ILP) model. The ILP lower bound provided by CBC [13], an open-

source MIP solver, was 600.167 in less than 1 CPU second. The optimal tour obtained in 24 CPU hours with an objective of 694 units is shown in Fig. 7.

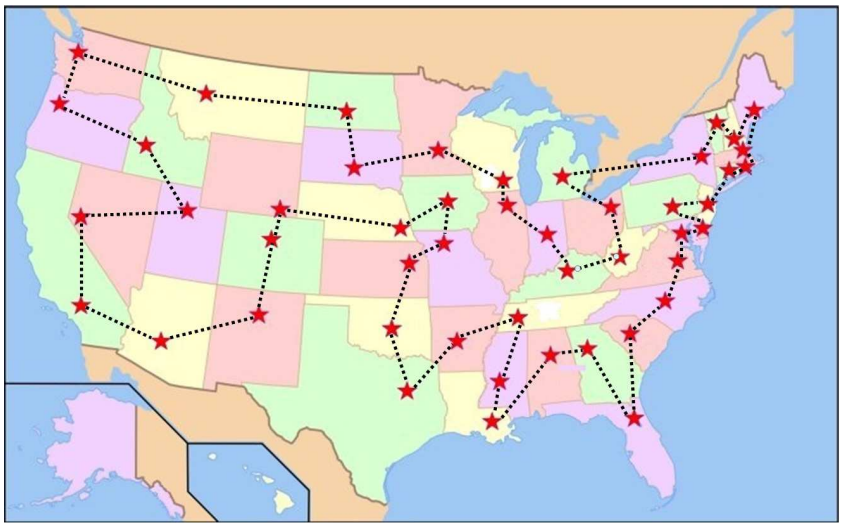

Figure 7 Optimal TSP tour of the 49-city instance

C. Network modification in GAMS

Given a TSP network, set N of original nodes, and a given base node, the task consists of removing any node and all of its corresponding arcs satisfying the following two conditions: (A) the node is within two arcs from the given base node, and (B) its degree, i.e., the number of edges incident to it, is less than or equal to δ. An example with $\delta = 5$ is provided on the left side in Fig. 6.

The following implementation solves the problem in GAMS.

```
$title Traveling Salesman Problem

set i cities / A, B, C, D, E, F, G, H, I, J, K, L, M, N,
               O, P, Q, R, S, T, U, V, W, X, Y, Z/;

$include 'TSPDataAndArcSets.inc'

* New sets, parameters & tables
set BaseNode(i) /O/;
alias (i,k,j), (i,ii);

parameter Degree(i) indicator function of incoming and outgoing arcs to node i,
          count;

* STEP 1: Get node degree values:
loop(i,
       count=0;
       loop(j$(i ne j),
             count$(c(i,j) gt 0)=count+1;
          );
```

```
       Degree(i)=count;
     );

* Counterpart Datalog's code:
* Degree[i] = v <- agg<<v=count()>> A[i,_]=_, city(i).

* STEP 2: Find nodes at most 2 arcs away from node base i of
*         degree less than or equal to 'delta':

set SubCity(i) subset of city nodes to be removed;

parameter alpha upper bound on nodes' degree / 5 /,
          FindRange2Paths(i,k,j) paths of range 2 from base nodes;

loop(i$BaseNode(i),
      loop(k$(k ne i),
          if( ( (c(i,k) ge 0) and (Degree(k) le delta) ),
            loop(j$(j ne i and j ne k),
                if( ( (c(k,j) ge 0) and (Degree(j) le delta) ),
                   FindRange2Paths(i,k,j) = c(i,k) + c(k,j);
                );
            );
          );
      );
    );

* Counterpart Datalog's code:
* Find:Range2Paths[i,k,j]=dij <- A[i,k]=dik, A[k,j]=djk, Degree[k]=dk, Degree[j]=dj,
                                  BaseNode(i), (dj<=alpha, dk<=alpha), dij=dj+dk.

* Next, we recover and remove all but the base node and the corresponding related arcs
* that are found on the paths.
* We accomplish this by making use of Dynamic Sets, i.e., subsets of underlying index sets.

set NewCity(ii) subset of allowed cities,
    R1(ii,jj) temporary subset of arcs in the network,
    R(ii,jj) final subset of allowed arcs in the network;
alias (ii,jj);

NewCity(ii) = city(ii).
* Make the arc set R1 complete; all arcs belong to the subset.
R1(ii,jj) = yes.

* Make a dynamic replica of the original TSP arc set 'TSParcs(i,j)'
* i.e., make R be the intersection of subsets TSParc and R1
R(ii,jj) = TSParcs(ii,jj)*R1(ii,jj);

* Then remove the required arcs and nodes from R and NewCity, respectively, and use them
* to construct the mathematical model.

loop((i,k,j)$(k ne i and j ne i and j ne k),
    if(FindRange2Paths(i,k,j) gt 0,
         NewCity(k) = no;
         NewCity(j) = no;
         R(k,j) = no;
         R(j,k) = no;
       );
    );

*Build compact TSP model
$include 'TSPModel.inc'

model MTZ /all/;

option optcr=0, reslim=30;
solve MTZ min z using mip;
```