Hewlett Packard Labs

The Specification for xLM: An Encoding for Analytic Flows

Alkis Simitsis, Kevin Wilkinson

Hewlett Packard Labs
HPE-2016-90

Keyword(s):
xLM; logical; physical; language; analytics; flows; optimization

Abstract:
We describe the xLM language for representing analytic flows. xLM (extensible Logical Model) encodes analytic flows that may span multiple engines and may have multiple constraints and objectives at flow and operator levels. xLM can be used for representing flows at logical and physical levels. It also covers control-flow logic and functional/non-functional requirements.

External Posting Date: October 25, 2016 [Fulltext]
Internal Posting Date: October 25, 2016 [Fulltext]

Hewlett Packard Enterprise

Technical Report

The Specification for xLM: An Encoding for Analytic Flows

Alkis Simitsis, Kevin Wilkinson

HP Labs, Palo Alto, California, USA

March 14, 2014
version 1.0

Abstract

We describe the xLM language for representing analytic flows. xLM (e**x**tensible **L**ogical **M**odel) encodes analytic flows that may span multiple engines and may have multiple constraints and objectives at flow and operator levels. xLM can be used for representing flows at logical and physical levels. It also covers control-flow logic and functional/non-functional requirements.

Table of Contents

1 Introduction

With the increasing capability of today's systems to process larger, and larger amounts of data, modern business intelligence (BI) and analytics have departed from the traditional architectures proposed and used in the 90's and 00's. In a modern enterprise, answering a business question may require a complex, analytic data flow that integrates datasets and computation from a number of diverse repositories and processing engines.

Conceptually, one may consider such a flow as a single, logical computation and it may be modeled as such. However, a logical flow has many possible implementations, each serving a different purpose. The job of the flow designer is to create an implementation (or physical flow) that meets objectives for the flow and workload. But, over time, objectives may change, data volumes may increase rendering an implementation sub-optimal, the underlying infrastructure may change, or the logical flow may need modification. Creating and modifying physical flows is labor-intensive, time-consuming, and error prone. Because enterprises are now deploying a wide variety of systems, such as Map-Reduce systems, stream processing systems, statistical analysis engines, and even elastic computing, the trend is toward more of these complex, hybrid analytic flows. And this increases the development and maintenance burden on IT departments.

What is needed is a notion of engine independence for logical analytic flows. Just as logical data independence insulates a data modeler from physical details of the relational database, there are benefits in designing flows at a logical level and using automation to implement the flows.

Towards this direction, we created a flow language, xLM, that encodes flows at both the physical and logical levels. With the appropriate tools (e.g., [2]), one may convert a flow that contains subflows written in n programming languages to a flow written in a unified form, e.g., xLM, and from there, to produce a semantically equivalent flow that contains subflows written in m programming languages.

In this document, we describe the xLM language. xLM is built in a way that eases the conversion of a physical flow constructed for one processing engine (engine-specific flow) to a logical flow that is engine independent (engine-agnostic flow) and vice versa. And to take it a step further, this eases the conversion of multi-engine, physical flows into a flow expressed as a unified, logical flow.

A logical, engine-independent flow gives an end-to-end view of the entire analytic computation. And this offers additional benefits.

First, it offers a means for translating a flow written in one language to a flow written in another language, through the engine-agnostic form of the flow using inter-engine language

translators. In other reports, we describe BabbleFlow, which is an example implementation of such a translator [1, 2]. Or, a flow processor might document the flow metadata either as pseudo-code or as a natural language description [3].

Second, an end-to-end view of the entire computation enables a number of possible and practical flow transformations. These may alter the flow design (e.g., flow structure) and physical properties to improve, for example, flow performance and/or flow recoverability, but not its semantics (functionality). In other papers, we present flow transformations like: (a) flow optimization [5, 6, 8]; (b) flow decomposition, which decomposes a single, large, complex flow into smaller subflows to reduce contention in a workload or to improve flow maintainability [7]; or (c) flow composition, which composes a series of individual, connected flows into one large flow to improve performance [7].

In addition to capturing structural and semantic information of a flow, as a flow graph describing an ordered series of operations along with their operational semantics, xLM has other advantages too. It also captures physical and logical properties and characteristics of a flow, like: physical resources used by the flow (e.g., memory and cpu budget, computation and storage paths); execution characteristics (e.g., operators' selectivity and throughput); design choices (e.g., parallelization, sorting, recovery points); implementation options (e.g., multiple physical implementations for a single logical operation); engine choices; operations' cost model, and so on. And beside flow design and execution details, xLM can also model business constraints related to a flow, like: preferable engine choices (e.g., process x% of the data on engine X and no more than y% of the data on engine Y); execution choices (e.g., run the process in time window W), fault tolerance constraints (e.g., use k replicas, use recovery points), and so on.

In what follows, we first present an example abstract flow in Section 2. In Section 3, we present a formalization for analytic flows. Then, in Section 4, we describe the xLM language constructs. In Section 5, we describe an implementation of a mappings dictionary. Finally, in Section 6, we discuss *xLM* extensions and more complex flow designs.

2 An Example Abstract Flow

Consider an analytic program that comprises two sub-programs running serially on two different engines, say E_1 and E_2, that the output of the first sub-program is consumed by the second, and that each sub-program performs a series of operations. This program can be seen as an analytic flow describing the computations needed as data flow from the input data stores toward the target data stores. Similarly, each sub-program can be seen as a subflow. As we formally discuss shortly (see sect. 3.1), an analytic program can be represented as a flow graph or simply a graph.

Figure 1 shows an abstract flow graph that represents an analytic program. This flow graph comprises two subflows (i.e., sub-programs) running serially on two engines, E_1 and E_2.

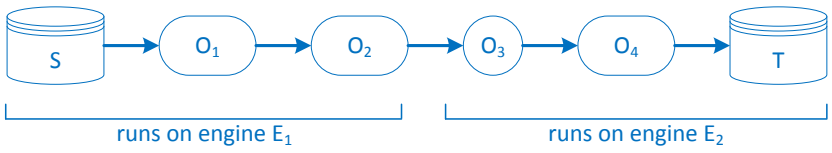

Figure 1: An abstract flow running on two engines

The flow starts with the O_1 operator, which reads data from a data store S (e.g., file, table, stream) and performs some computation (e.g., filter, lookup, sentiment analysis, clustering, feature extraction from an image/video/audio file, etc.). Then, it sends the data to another operator, O_2, that performs a different computation. The O_1 and O_2 operators are executing on the E_1 engine. Next, the data are shipped to another engine, E_2, through a *connector* operator, O_3. Then, the computation continues with the O_4 operator on engine E_2, and the result data are stored in a target data store T.

The two data stores S and T may use persistent storage or not, may be be partitioned or not and may be located on the same or different physical machines, in the same or different file systems. For example, S may be stored in a Map-Reduce engine, e.g., in HDFS for a Hadoop implementation, and T may be a table in a database engine.

The connector operator can be implemented in several ways: it may run on either engine pulling or pushing data from one engine to another, or it may be a process running outside the two engines (e.g., a shell script running in the operating system) and orchestrating the data move. Note, that in Figure 1, we use a different shape for the connector operator for presentation reasons.

Abstracting out such implementation details, allows us to derive a logical model for the flow, which is engine independent. Such a logical model enables handling the end-to-end flow as a unified entity, as discussed in Section 1. As an example flow transformation, we could for example consider function shipping and move operator O_4 to engine E_1 and/or operator O_2 to engine E_2. A more detailed analysis of such flow transformations can be found elsewhere (e.g., [6]).

In the next sections, we first describe how we can formally model and encode an analytic flow with *xLM*. After having described the core design constructs, we also discuss more advanced flow designs involving parallelism, control-flow functionality, and business requirements.

3 Flow Formalization

In this section, we describe a formalization for analytic flows and describe how it can be used to support a generic form of flows written in different languages. The casual reader may skip this section and move forward to Section 4 for a description of *xLM* constructs.

3.1 Preliminaries

We model an analytic flow Γ as an acyclic, parameterized digraph $\Gamma(U_\Gamma)=(V_\Gamma(U_\Gamma), E_\Gamma)$, where U_Γ is a finite set of properties of the vertices V_Γ of Γ. V_Γ are either operators, V_{op}, or data stores, V_{ds}. A vertex may be assigned to an engine; if it is an operator, then it runs on that engine, if it is a data store, then it stores data on that engine. The edges E_Γ model the data flow among the vertices. A special class of operators is the connectors, $V_{cn} \subset V_{op}$, which connect parts of the flow that run on different engines. The vertex properties capture information related to business requirements, Q, resource allocation, R, and characteristics, $\mathcal{C}$, like the vertex type $\mathcal{C}_{type}$ (e.g., sentimentMiner, join), implementation type $\mathcal{C}_{impl}$ (e.g., merge-sort join), engine used $\mathcal{C}_{eng}$ (e.g., Hadoop, database), etc. Therefore, the properties of a vertex v_j in Γ are $U^j_\Gamma = Q^j_\Gamma \cup R^j_\Gamma \cup \mathcal{C}^j_\Gamma$.

In general, the vertices of an analytic flow Γ may be assigned to a multiplicity of engines. The set of all engines used in Γ is represented as $\Phi_\Gamma = \bigcup_j \mathcal{C}^j_{\Gamma eng}$, for all vertices $v_j \in V_\Gamma$. Connected vertices assigned to the same engine constitute a subflow of Γ. Hence, Γ may comprise a partially ordered set of such subflows $I_K=\{G_i\}$, $1 \leq i \leq K$ (K being the size of the set), each one having vertices assigned to a single engine. Each of these subflows constitutes an acyclic digraph $G_i=(V_i, E_i)$. Their partial order in Γ is defined by the reachability of the flow. The reachability relation of Γ is the transitive closure of its edges set E_Γ, i.e., the set of all ordered pairs (x, y) of its vertices in V_Γ for which there exist vertices $v_1=x$, ..., $v_{|V_\Gamma|}=y$, such that $(v_{j-1}, v_j) \in E_\Gamma$, for all $1 < j \leq |V_\Gamma|$. Depending on the structure of Γ, we have three types of flows:

(Single flow) A analytic flow is a *single flow* iff $|I_K|=1$.

(Multi-flow) A analytic flow is a *multi-flow* iff $|I_K| > 1$.

(Hybrid flow) A multi-flow is a *hybrid flow* iff $|\Phi_\Gamma| > 1$.

Note, that each of the K subflows of a multi-flow is called a single flow. An analytic flow itself may be a single flow too.

In what follows, we use the terms flow and graph interchangeably. We do the same for single flow and subflow.

3.2 Different forms of a flow

A flow may be seen at different levels.

A *logical* flow is independent of an execution engine. A graph G_L representing such a flow contains vertices that do not necessarily have resource allocation information or some of their characteristics completed (e.g., implementation type).

A *physical* flow is a flow that is assigned to a specific execution engine (or engines, in the case of hybrid physical flows); specific execution engines; e.g., specific RDBMS, specific map-reduce engine, specific ETL engine. A graph G_P representing such a physical flow contains the information needed to bound an operator for example to a specific implementation and engine.

A flow running on an engine is expressed in a *language*, L, that the engine can execute. This language may be programming code or even metadata that the engine interprets. Given n languages, we would need $n \times (n-1)$ parsers to convert one language to another. We follow a different approach: we introduce an intermediate language, L_N, that all other languages should be converted to first. Hence, we reduce the number of parsers needed to $2 \times n$. A previous use of this idea goes back to the early days of NL processing [4].

L_N has the following characteristics. It describes flows, their operators (schemata, semantics) and the interconnection among them. It captures additional operational properties at the flow and operator levels like resources required and physical characteristics. It can also represent various levels of abstraction. We use L_N as our logical flow language. Hence, a single, physical flow expressed in a language L_i is translated first to L_N and from there, it can be converted back to the same or to another language.

In the rest of this manuscript, we present xLM, which is an implementation of L_N. This implementation, as we describe shortly, allows keeping logical constructs in L_N and also L_i constructs, i.e., engine specific details for multiple engines. Depending on whether xLM encodes a logical flow G_L or a physical flow G_P, we call it engine agnostic xLM (denoted as *a-xLM*) or engine specific xLM (denoted as *s-xLM*), respectively. As we discuss shortly, we may keep both *a-xLM* and *s-xLM* elements in the same xLM representation.

4 Language Constructs

We describe here an xLM implementation in terms of XML encoding. Other encoding schemes could be used too, like JSON.

We separate the xLM constructs in three categories. The first, the *core elements*, contains the main structural components of a flow representation. The second, the *constructive elements*, contains elements that are used as atomic entities in the construction of more complex elements. The last one, *physiological elements*, involves physiological characteristics of flow components, which represent various logical and physical characteristics of a flow.

4.1 Core Elements

The main building blocks of a flow in xLM include: design, which encodes a flow and is used as a container for all other elements; metadata, which keeps data useful to interpret, execute, and translate a flow from one form to another, but it does not necessarily contain structural information of the flow; edges, which describe the flow of data among the flow nodes; and nodes, which describe the structural and operational semantics of the flow operators and data stores.

4.1.1 Design

Design (<design>) is the top-level component of xLM and it should be seen as the equivalent of a flow. All other language constructs are contained in <design>. An abstract structure of <design> is as follows:

```
<design>
    <metadata/>
    <edges/>
    <nodes/>
    <properties/>
    <resources/>
    <features/>
</design>
```

4.1.2 Metadata

Metadata (<metadata>) stores useful information for a flow. Typically, this information does not describe flow structure, constraints, or objectives; these are captured by the elements we describe shortly. Metadata stores physical details for a flow that are not required by a flow processor, like specific details needed for converting a flow from one form to another. For example, metadata may store details needed for converting a flow

from an engine-specific form to an engine-agnostic form (e.g., from SQL to xLM) and vice versa (e.g., from xLM to SQL). An abstract structure of <metadata> is as follows:

```
<metadata>
    <name/>
    <info/>
    <flowID/>
    <creationDate/>
    <xlmVersion/>
</metadata>
```

Metadata stores the name (<name>) of a flow and a generic info (<info>) element. <info> keeps the physical details for a flow implementation as CDATA. This may be for example boilerplate code needed by a specific engine to execute the flow, but this code does not contain flow semantics required to interpret flow functionality. Also, a flow may contain subflows. Each subflow may have a separate entry in <info>. For example, the metadata of a flow named 'myFlow' that contains two subflows named 'flow1' and 'flow2' can be stored as follows:

```
<metadata>
    <name>myFlow</name>
    <info>
        <flow1><![CDATA[ info for flow1 ]]></flow1>
        <flow2><![CDATA[ info for flow2 ]]></flow2>
    </info>
</metadata>
```

Metadata also store additional generic flow information like flow ID, creation/modification date/time, xLM version, etc.

4.1.3 Edges

Edges (<edges>) represent the data flow among the flow nodes, i.e., operations and data stores. <edges> comprises a set of edge elements (<edge>). An abstract structure of <edges> is as follows:

```
<edges>
    <edge/>
    <edge/>
    ...
</edges>
```

An edge, <edge>, represents the connection between two nodes. Only one edge is allowed between two nodes. An <edge> has a source (<from>) and a target elements (<to>), capturing the source and target node names, respectively. Optionally, an edge can be also extended with additional properties, like isEnabled (<enabled>), color (<color>), dashtype (<dashtype>), width (<width>), etc. An example <edge> is as follows:

```
<edge>
    <from>operator1</from>
    <to>operator2</to>
    <enabled>Y</enabled>
    <color>black</color>
    <dashtype>solid</dashtype>
</edge>
```

4.1.4 Nodes

Nodes (<nodes>) represent flow nodes like operations and data stores. <nodes> comprises a collection of node elements (<node>). An abstract representation of nodes is as follows:

```
<nodes>
    <node/>
    <node/>
    ...
</nodes>
```

A node, <node>, contains the information needed to characterize a flow operation or data store at either the logical or physical levels. A node contains the following elements:

<name/>	The name of the node instance. The <name> of a node should be unique among all nodes of a flow.
<type/>	The type of the node in the flow; i.e., Operator, Datastore or Intermediate node. (Intermediate is a special case of a data store node; e.g, it can be used as a temp or a check point.)
<optype/>	The operational type of a node. optype represents the logical type of the node; i.e., FileInput for a data store or Filter for an operator
<implementation/>	The physical implementation of the node. A single logical operator may have multiple physical implementations; e.g., Merge or Hash for a join operator.
<engine/>	The engine chosen for running this node. For engine-agnostic flows, the engine is set to xlm, otherwise the name of the engine is used.
<flowID/>	The subflow identifier this node belongs to. If the flow is a single flow, then this is the flow id. If the flow is a multi-flow, then this element shows the flow id of the subflow this node is assigned to.
<schemata/>	The schemata of this node (see sect.4.2.1).
<ndproperties/>	The properties for this node (see sect.4.3.1).
<ndresources/>	The resources for this node (see sect.4.3.2).
<ndfeatures/>	The features for this node (see sect.4.3.3).

4.2 Constructive Elements

Here, we describe two elements: schemata and expressions. They are both used as atomic constructs for building more complex elements.

4.2.1 Schemata

An analytic flow represents the flow of data from one node to another. The layout of the data stored on a node or passing through a node is represented by the schemata (<schemata>) of the node. Different nodes have different schemata. An operator node has input, output, and parameter schemata. The first two represent the schema of data that this operator gets at its input and produces at its output, respectively. The parameter schema represents the parameters needed by this operation. A datastore or an intermediate node has only one schema; without loss of generality, we consider it as the input

schema. Hence, <schemata> has the following generic structure:

operator nodes:

```
<schemata>
    <input/>
    <output/>
    <parameter/>
</schemata>
```

datastore/intermediate nodes:

```
<schemata>
    <input/>
</schemata>
```

Input and output. Input and output schemata contain a collection of attributes (<attr>) and an identifier of their multiplicity (<card>), as follows (we show input, but output is the same too):

```
<input>
    <card/>
    <attr/>
    <attr/>
    ...
</input>
```

Therefore, if a node has more than one schema of the same category, e.g., two input schemata (similarly for output), then we use <card> to separate them:

```
<schemata>
    <input>
        <card>1</card>
        <attr/>
        ...
    </input>
    <input>
        <card>2</card>
        <attr/>
        ...
    </input>
</schemata>
```

Each <attr> describes an attribute of a schema as follows:

```
<attr>
    <name/>
    <type/>
    <attrproperties/>
    <ifnull/>
</attr>
```

Attribute <name> and <type> captures the name (e.g., pKey) and type (e.g., Integer) of an attribute, respectively. Additional properties are captured by <attrproperties>. A special case is the ifNull property <ifnull/> that stores a predefined value for the attribute in case its value is $null$.

Parameter. The parameter <parameter> schema is valid only for operator nodes. It describes the parameters needed for executing an operator. At the logical level, this is the only information available to characterize the operational semantics of the operator.

A <parameter> element contains a collection of <param> elements. Each <param> describes a single expression. <param> is engine-specific information and it should be repeated for every engine-specific implementation considered for a given node. A parameter schema can be as follows:

```
<parameter>
    <param/>
    <param/>
    ...
</parameter>
```

A <param> element contains the following:

<pengine/>	The type of the engine that this parameter is intended to run onto.
<ptype/>	This element specifies which part of an operator's code the expression <expr> refers to.
<expr/>	The expression of the parameter (see sect.4.2.2).

Valid values for <pengine> and <ptype> are extracted dynamically from the dictionary (see sect. 5). Example engines may be a database or a Map-Reduce engine. For flows that run on a database, xLM encodes the flow characteristics in a SQL-specific form. Example <ptype> values for SQL might be where, select, groupby, and so on.

An example parameter schema for a <Filter> operator, with engine-agnostic (xlm) and engine-specific (here sql, for a database engine) implementations, is as follows:

```
<parameter>
    <param/>
        <pengine>xlm</pengine>
        <ptype>filter_cond</ptype>
        <expr>...</expr>
    </param>
    <param/>
        <pengine>sql</pengine>
        <ptype>where</ptype>
        <expr>...</expr>
    </param>
</parameter>
```

In this example, the xlm implementation can be used to convert a parameter expression from/to an engine-specific implementation. The sql implementation can be used to generate SQL code for this specific expression, which in this example case will be translated to a predicate in a WHERE clause of a SQL statement. (Deciding in which WHERE clause if there are multiple or in what predicate order within the WHERE clause depends on the semantics of the node and the flow.)

4.2.2 Expressions

Expressions are extensively used in xLM for constructing various elements like parameters, properties, resources, and features. Essentially, expressions model a complex key-value pair relationship, where both key and value may come as a result of a computation (e.g., function) and their relationship may be defined by an arbitrary operator.

An expression is represented as an <expr> element, which defines the left and right operands, functions over them, and an operator that interconnects them. Therefore, to represent an expression of the form:

$$f(x) \diamond g(y) \tag{1}$$

where x and y are the left and right operands, f and g are functions applied to these operands, and $\diamond$ is an operator, we use the following elements:

<leftfun/>	A function applied over the left operand.
<leftop/>	The left operand.
<op/>	An operator.
<rightfun/>	A function applied over the right operand.
<rightop/>	The right operand.

An operand may represent a variable or a constant value. Also, it may be simple, i.e., a single variable or a single value, or complex, i.e., a combination of variables and/or values. Based on this distinction, we have two expression categories.

Simple expressions. A simple expression is of the form shown in the equation (1), where x and y are simple –single– operands. This is a common form used to capture most of the flow or node characteristics as in properties, resources, or features. For example, the selectivity of a flow operator and the value for a startDate field can be captured with the following expressions:

selectivity example:

```
<expr>
    <leftfun></leftfun>
    <leftop>selectivity</leftop>
    <op>=</op>
    <rightfun></rightfun>
    <rightop>1</rightop>
</expr>
```

date example:

```
<expr>
    <leftfun></leftfun>
    <leftop>startDate</leftop>
    <op>=</op>
    <rightfun></rightfun>
    <rightop>
        getDate('2013-11-13','YYYY-MM-DD')
    </rightop>
</expr>
```

Note that in the data example the getDate function returns a value and thus it is captured by the rightop element. In the next paragraph, we describe an example where a function is applied on a field and thus, the function is captured by the the rightfun element and the field by the rightop element.

Complex expressions. A complex expression follows the form of equation (1), but the operands x and y may represent more than one variable. In addition, the functions f and g may represent a more complex computation either on a single or multiple variables. In a complex expression, we use a template mechanism to represent the intended computation.

Templates can be used in function elements. A template is identified with $$ at the beginning of a function element content. In a template, a special character $ followed by a numerical identifier work as a placeholder for a variable name. The list of variables to be used in a template is stored in the respective operand element content. The order of placeholders in a template is the identical to the order in which variables in an operand element are listed.

For example, the predicate:

ToDate(startDate,'YYYY-MM-DD') < ToDate('2013-11-13','YYYY-MM-DD');

where startDate is an attribute name, is represented by the following xLM snippet:

```
<expr>
    <leftfun>$$ ToDate($1,'YYYY-MM-DD')</leftfun>
    <leftop>startDate</leftop>
    <op>&lt;</op>
    <rightfun></rightfun>
    <rightop>
        ToDate('2013-11-13','YYYY-MM-DD')
    </rightop>
</expr>
```

Multiple variables in a template can be used as follows:

```
<expr>
    <leftfun></leftfun>
    <leftop>salary</leftop>
    <op>=</op>
    <rightfun>$$ $1 + $2</rightfun>
    <rightop>wages, bonus</rightop>
</expr>
```

Instantiating the template in this snippet will result into: salary = wages + bonus.

Expressions in parameters. xLM uses expressions to encode predicates, computations, value assignment, etc. Interpretation of value assignments as in properties, resources, and features or computations is straightforward. However, in parameters, expressions can be used to encode predicates that can be placed in various parts of an operator's code.

Let's consider as an example, the filter operator discussed in 4.2.1-Parameter with the predicate on dates that we just described. The code listing in Figure 2 illustrates how this predicate can be captured in a parameter schema for multiple implementations: for xLM, SQL, and Apache PigLatin (hereafter, PigLatin).

In this example, the expression encodes a predicate of the filter operation. In SQL, this is part of the WHERE clause of the query. In PigLatin, it is part of a BY clause. And in xLM, it is captured as part of a filter condition (filt_cond).

The attribute used in a parameter schema must belong to the input schema of the respective operator. For example, in Figure 2, the date function is applied over the startDate attribute, which should belong to the input schema of the filter operator.

```
<parameter>
    <param/>
        <pengine>xlm</pengine>
        <ptype>filter_cond</ptype>
        <expr>
            <leftfun>$$ (getDate($1,'YYYY-MM-DD'))</leftfun>
            <leftop>startDate</leftop>
            <op>&lt;</op>
            <rightfun></rightfun>
            <rightop>(getDate('2013-11-13','YYYY-MM-DD'))</rightop>
        </expr>
    </param>
    <param/>
        <pengine>sql</pengine>
        <ptype>where</ptype>
        <expr>
            <leftfun>$$ to_date($1,'YYYY-MM-DD')</leftfun>
            <leftop>startDate</leftop>
            <op>&lt;</op>
            <rightfun></rightfun>
            <rightop>to_date('2013-11-13','YYYY-MM-DD')</rightop>
        </expr>
    </param>
    <param/>
        <pengine>pig</pengine>
        <ptype>by</ptype>
        <expr>
            <leftfun>$$ ToDate($1,'YYYY-MM-DD')</leftfun>
            <leftop>startDate</leftop>
            <op>&lt;</op>
            <rightfun></rightfun>
            <rightop>ToDate('2013-11-13','YYYY-MM-DD')</rightop>
        </expr>
    </param>
</parameter>
```

Figure 2: Example parameter schema for a filter condition in *xLM*, SQL, and PigLatin

As an aside note, observe how *xLM* captures the different versions of the date function (getDate, to_date, ToDate) in different languages (more in sect. 5).

Similarly, an expression can be used to represent a built-in function or a user-defined function (udf) at various parts of an operator's code. And expressions can also be used for giving identifiers or aliases to attributes or computations over attributes.

For example, in the same filter operator, we can augment the parameter schema by adding a new <param> for representing a function, e.g., sum, applied on one of the attributes projected by the operator. The following snippet shows an implementation of this in SQL. When evaluated it will generate: SUM(qty) as sumQty, which will be added in the SELECT clause of the SQL query. Note also the assignment operator :=.

```
<param/>
    <pengine>sql</pengine>
    <ptype>select</ptype>
    <expr>
        <leftfun></leftfun>
        <leftop>sumQty</leftop>
        <op>:=</op>
        <rightfun>$$ (SUM($1))</rightfun>
        <rightop>qty</rightop>
    </expr>
</param>
```

4.3 Physiological Elements

The physiological elements represent various logical and physical characteristics of a flow, namely flow and/or node <properties>, <resources>, and <features>. Each of these can be used at two levels: at the <design> level and at the <node> level. Depending on which level they are used at they may have a different name; for example, properties at the design level are represented as <properties> and at the node level as <ndproperties>. We use the prefix nd similarly for <resources> and <features>. However, the internal structure of these elements is the same at the both levels. We elaborate on this next.

4.3.1 Properties

Properties store characteristics of the design. These are defined in the dictionary and used by xLM parsers to store flow/node metadata. For example, we can store as properties the

file_path or uri etc. of a data store; or storage_type like org.apache.pig.builtin.PigStorage for PigLatin storage.

Properties, either <properties> or <ndproperties>, are collections of property elements, <prop> or <ndprop>, respectively. A <prop> or an <ndprop> have two elements: <ptype> and an expression, <expr> (see sect. 4.2.2).

Recall from section 4.2.1 that the schemata of a node have a <card> element that specifies their multiplicity. An n-ary node has n input schemata and thus, <card> may take values from 1 to n. To keep track of which node populates each of the input schemata of an n-ary node, we use a special property for nodes: card_origin. Consider for example that a node Group feeds the second input schema of a node MultiJoin. This input schema would have <card>=2. To encode this relationship, MultiJoin would have the following node property:

```
<ndprop/>
    <ptype>card_origin</ptype>
    <expr>
        <leftfun></leftfun>
        <leftop>2</leftop>
        <op></op>
        <rightfun>/rightfun
        <rightop>Group</rightop>
    </expr>
</ndprop>
```

In this example, the <leftop> element encodes the cardinality of MultiJoin's input schema and <rightop> encodes the name of the node that feeds that schema.

4.3.2 Resources

Resources store physical characteristics of a flow (<resources>) or a node (<ndresources>). Example resources include selectivity, throughput, dataset size, memory budget (i.e., how much memory a flow or a node can use before data spill to the disk), and so on.

Resources, either <resources> or <ndresources>, are collections of resource elements, <resource> or <ndresource>, respectively. A <resource> or an <ndresource> has two elements: <ptype> and an expression, <expr> (see sect. 4.2.2). Most often, resources are expressed as key-value pairs of ($resource\_name$,$resource\_value$), and thus the expression is simple:

```
<ndresource/>
    <ptype>resource_name</ptype>
    <expr>
        <leftfun></leftfun>
        <leftop>resource_name</leftop>
        <op>=</op>
        <rightfun>/rightfun
        <rightop>resource_value</rightop>
    </expr>
</ndresource>
```

In such cases, <ptype> content is the same as in <leftop>: $resource\_name$.

4.3.3 Features

Features store graphical information for drawing the flow graph in an xLM GUI. Example features include the coordinates of a flow or a node (xloc and yloc), node colors, shadow, shape, size, borders, bounding box, etc.

Features, either <features> or <ndfeatures>, are collections of feature elements, <feature> or <ndfeature>, respectively. A <feature> or an <ndfeature> has two elements: <ptype> and an expression, <expr> (see sect. 4.2.2). Most often, features are expressed as key-value pairs of ($feature\_name$,$feature\_value$), as follows:

```
<ndfeature/>
    <ptype>feature_name</ptype>
    <expr>
        <leftfun></leftfun>
        <leftop>feature_name</leftop>
        <op>=</op>
        <rightfun>/rightfun
        <rightop>feature_value</rightop>
    </expr>
</ndfeature>
```

Again, in such cases, <ptype> content is the same as in <leftop>: $feature\_name$.

5 Dictionary

A logical operator may have multiple physical implementations either on a single engine or across multiple engines. For example, a join operator can be implemented as a nested loop or hash join in a database and as a replicated or skewed join in PigLatin.

In order to achieve engine inter-operability and preserve flow semantics across multiple engines, we need a means for translating engine specific characteristics from one engine to another. To deal with this, we describe a *dictionary* of mappings.

In addition to keeping information useful for code interpretation and generation, the dictionary also contains attributes that can be used during flow processing, like the operator cost models specific to an implementation and engine.

The dictionary comprises: (a) categories; (b) language specific mappings; and (c) operation mappings.

Categories describe in a machine-processable way the dictionary structure and mapping types. The mappings connect the different incarnations of flow constructs for multiple engines. Categories are used as an index and allow changing the dictionary at runtime without affecting our system's operation. Figure 3-left shows example categories, like operator types (<optype>), boolean operators (<boolop>), and math operators (<mathop>); but many others are available and new categories can be added too.

For preserving flow semantics across engines, we handle different data types, expressions, and operators with the language specific mappings in physical to logical conversion and the operation mappings in logical to physical translation.

The language specific mappings are engine specific, as they capture the intrinsic characteristics of an engine and map them to engine agnostic *xLM*. Many engines separate the logical operator names from the internal, physical name corresponding to a specific implementation. For example, PigLatin FILTER translates into LOFilter when the script code translates into an execution plan; see also the pls_ls_mappings in Figure 3-left. There is also a variety of representations for functions and operands used in expressions across engines; e.g., the function ROUND is invoked in PigLatin as a call to a library (org.apache.pig.builtin.ROUND). Differences also occur among data types, and thus, we convert engine specific data types to logical data types.

The operation mappings describe the logical operations supported; e.g., operator and data store types. Figure 3-left, in the middle, shows a placeholder for operation mappings.

Figure 3-right shows an example entry in the dictionary for a FILTER operator, which has multiple implementations per engine and across engines. An operator entry has associated

```
{
   "categories":[
      {"cat":"optype"},
      {"cat":"boolop"},
      {"cat":"mathop"},
      …
    ],
   "operations":[
      …
      {"category" : "optype",
       "xlm.name" : "Filter",
       "cost": "xlm.FILTERCOST",
       …
      },
      …
    ],
   "pig_ls_mappings":{
      …
      "LOFilter":"FILTER",
      …
      "org.apache.pig.builtin.ROUND":"ROUND",
      …
      "GreaterThan":">",
    } …
}
```

```
{"category" : "optype",
 "xlm.name" : "Filter",
 "pig":[{"name" : "Filter",
          "ptype":[{"filter_cond" : "BY"}],
          "cost":"pig.FILTER"
          }],
 "sql":[{"name" : "Selection",
          "ptype":[{"filter_cond" : "WHERE"}],
          "cost":"sql.SELECTROWS"
          }],
 "pdi":[{"name" : "FilterRows",
          "ptype":[{"filter_cond" : "condition"}],
          "impl":"FilterRows",
          "cost":"pdi.FILTERROWS"},
        {"name" : "JavaFilter",
          "ptype":[{"filter_cond" : "condition"}],
          "impl":"JavaFilter",
          "cost":"pdi.JAVAFILTER"}]
}
```

Figure 3: Example entries in the dictionary

attributes, including: a logical operator name (e.g., `'xlm.name'='FILTER'`), a link to a cost model for computing the operator's cost, and template structures for the translation of the operator to a physical implementation. Figure 3-right shows example implementations: in PigLatin, `Filter`; in SQL, `Selection`; and in Pentaho PDI, an open source ETL tool, two implementations of filter, `FilterRows` and `JavaFilter`. The physical implementation details can be used in code generation or by a flow processor; e.g., for choosing the appropriate cost model for an operator. Other attributes stored for operators include links to code templates or implementation specific, physical properties like whether an operator is order preserving, parallelizable, streaming, and so on.

The dictionary can be implemented in various ways. Figure 3 shows an example implementation, where the dictionary is implemented as a single file in JSON format. Other formats are straightforward to use.

Modifying the dictionary is a semi-automated process. The dictionary is extensible to new engines and implementations. For adding a new language or modifying an existing one, a template dictionary instance can be used. When one finishes entering the details for the new/updated language, then an automated mechanism can be used for updating the dictionary accordingly.

6 Extensions to xLM

So far, we described how *xLM* encodes the main components of analytic flow at the logical and physical levels. Now, we discuss how *xLM* encodes additional flow functionality and structures, like parallel subflows in a flow, control-flow logic, and business requirements.

6.1 Parallel flows

Operators and subflows within a flow may be placed and run in parallel branches for various reasons. Here, we focus on two example scenarios: partitioning and replication. As far as *xLM* is concerned, these two cases are handled similarly. The same holds for other scenarios of having parallel branches in a flow design.

Flow partitioning. There are several ways to represent parallelization in flows. Here, we describe an example one. It is straightforward to use *xLM* to encode other forms of parallelization too.

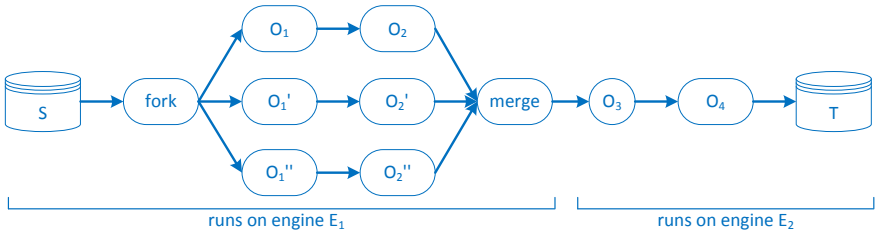

Figure 4: Example abstract flow with parallelization

Figure 4 shows a variant of the abstract flow of Figure 1, where part of the computation performed on engine E_1 is done in parallel[1]. At the logical level, a full-fledged version of an *xLM* encoding for such a flow, involves adding two new operators, fork and merge, for splitting and then, merging the data flow. Each operator instance that runs in parallel, creates a new <node> in the <design>. In general, the nodes representing parallel operator instances have identical elements with the exception of the node <name>. Based on the partitioning method used, other elements may be different too, like the parameter schema of the nodes, <parameter>. For example, for range partitioning, operator's <parameter> should also reflect a filter on the range of data values that is of interest for each specific operator instance. Similarly, for hash partitioning or other partitioning types.

[1]Here, we assume that both operators have a parallelizable implementation. In *xLM* this can be captured as a node property, <ndprop>, namely isParallelizable. A more detailed discussion on correctness issues is out of the scope of this document.

At a physiological level, parallel operator instances may differ in other elements too, especially in those that capture physical information like resource allocation; e.g., buffer size, data path, storage location, execution path (parallel flow branches may run on different environments, like different physical or virtual machines, etc.). In general, the elements affected belong to <ndproperties> and <ndresources>.

Depending on the operator type, the operator implementation, <implementation>, may change too. For example, assume that operators O_1 and O_2 of Figure 1 are a generic aggregator and a generic filter, respectively. The filter is not a holistic operator (i.e., it does not need to process all the input data, before it outputs the first result tuple) and thus, it can be easily parallelized. In this case, the three instances of O_2 in Figure 4 may have the same implementation type. The case of the aggregator might be different depending on the nature of the aggregation and the implementation used. If O_1 is a holistic operation, we may need extra functionality to retain the semantics intended. For example, for a simple grouper, we could do sort in parallel, and then add a merge sort to combine the results. This would alter the flow design and the *xLM* encoding for the flow would alter as needed. But interestingly, the physical and, often, complicated details of such an alteration are encapsulated in a simple change of only one element: <implementation>.

It is worth noting that partitioning is one way to achieve parallelism. Another way is to consider pipeline parallelism. Pipeline parallelism is handled at the physical level. In *xLM*, this may be captured as <ndresources> to indicate either that an operation does support pipeline parallelism or that runs in pipeline fashion. This information can be then used by a flow processor (e.g., a flow optimizer).

Flow replication. Flow replication is relevant when a designer wants to replicate subflows within a flow (or as an extension, to replicate the entire flow) for example, for fault tolerance. In this context, one could read the flow in Figure 4 as a flow where we use triple modular redundancy (TMR) for the subflow running on engine E_1. In such a case, the merge operation could be implemented as a voter that looks at the results of the three branches and picks the correct one based for example on majority voting. In this scenario, the operator instances in the parallel branches are identical (same implementation, schemata, etc.), with only one exception: their names (each operator in a flow should have a unique name).

Notation issues. As a syntactic sugar, it is not necessary to represent a subflow that runs in parallel branches as shown in Figure 4 for the operators O_1 and O_2. A GUI for *xLM* flows could hide the parallel branches and also, encapsulate the fork and merge operators to the operators placed at the edges of that subflow.

An example representation might be as follows. Every operator that runs in parallel could be annotated with a symbol representing parallel execution (this symbol could also show

the number of parallel branches involved). The fork and merge functionality could be added to the operators or data stores where the parallel subflow reads from and writes to, respectively, data. A simple way to achieve this is by using additional input and output schemata for those operators as needed.

However, if the fork or merge operations use complex semantics like a special type of partitioning or merging, then, it is advisable to use separate operators as shown in Figure 4. Having operators that perform as simple tasks as possible would help a flow processor to better transform a flow; e.g., separating a data store node and a fork operator would allow a flow optimizer to move a selective operator between the data store and the fork and this could potentially minimize the amount of data need to be partitioned. On the other hand, having composite operators leads to more readable designs. *xML* allows both, as clearly, this is a tradeoff for a flow designer to handle based on the objectives at hand.

6.2 Control flows

The discussion so far described design aspects for analytic data flows. However, *xLM* is generic and extensible enough to support control flow functionality as well. To facilitate the discussion, we separate data flow operations from control flow operations as follows. A data flow represents the flow of data from the source to the target data stores. A control flow describes how data flows are interconnected referring to the order in which the data flow operators are executed or evaluated. This separation is not novel; it is a rather common technique in flow engines like workflow or ETL engines, where there is a separation between jobs and tasks (different tools use different names as stages and jobs, jobs and transformations, and so on).

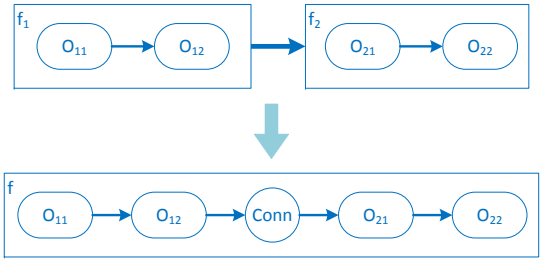

Figure 5: Example control and data flows

The top of Figure 5 shows an excerpt from an example, conceptual control flow that connects two data flows f_1 and f_2. Each data flow comprises a series of data flow operators: O_{11} and O_{12} for f_1 and O_{21} and O_{22} for f_2. In *xLM*, the entire flow can be seen as one entity as shown in the bottom of Figure 5: the *xLM* encoding is produced as a flattened version of the control flow.

To represent control functions, we use connector operators. Example control functions include: conditional operators (e.g., continuing the flow only if some condition is met), block operators (e.g., halt data flow until a condition is met), etc. Such conditions can be described in the <parameter> schema of a connector. Additional issues to be taken under consideration include types of data movement from one data flow to another; e.g., pipeline data, use an external (temp) storage, move data only after a condition is met, etc. These issues are captured by the <parameter> schema, but also by the <optype> and <implementation> of the <node> describing a connector operator.

For provenance purposes, when we convert a control flow to its flattened version in *xLM*, we annotate its <node> with the flow identifier of the original data flow that it comes from. We use the <flowID> element for this matter (see sect. 4.1.4). We also keep any extra information needed to reconstruct the original control flow from its flattened *xLM* version in the design metadata, in the <info> element (see sect. 4.1.2).

Appropriate parsers can be used to connect flow engines to *xLM* in order to enable conversion of a control flow to *xLM* encoding as shown in Figure 5 and then, (possibly after flow transformation) back to a control flow able to execute on a flow engine (the reverse direction of the one illustrated in Figure 5).

6.3 Functional and non-functional requirements

Often, analytic flows come with business requirements that need to be satisfied. Example requirements include: “the flow should run every 2 minutes”, “the subflow reading data from source A should run every 1 minute and the subflow reading data from source B should run every 10 minutes”, “the latency in updating target B should be at most X”, “the flow should tolerate up to K failures”, and so on.

These requirements are typically formed as service level agreements (SLAs) and are expressed in various forms, either structured or unstructured (e.g, as natural language descriptions). We do not elaborate here on how business requirements can be captured or expressed. Without loss of generality, we consider that such requirements can be formed in a structured way (e.g., as i* profiles [10]).

xLM can capture requirements at both the <design> and <node> levels. At the design (or node) level, both <properties> (or <ndproperties>) and <resources> (or <ndresources>) can be used, depending on the type of requirement. For example, we could encode a requirement on latency in updating a data store as a function of network congestion as a node resource for a node representing the data store involved and write:

```
<ndresource/>
    <ptype>requirement</ptype>
    <expr>
        <leftfun></leftfun>
        <leftop>latency</leftop>
        <op>=</op>
        <rightfun></rightfun>
        <rightop>cmpt_net_cong()</rightop>
    </expr>
</ndresource>
```

In general, using expressions (see sect. 4.2.2), *xLM* can encode fairly complicated requirements that may span various aspects of the flow design and execution.

7 Acknowledgments

The first discussions about the need for an extensible data flow language started at HP Labs (HPL) in late 2009, as we were working on the problem of handling business processes, extract-transform-load (ETL) design, and online analytical processing (OLAP) queries as a single, unified process spanning multiple levels of design: conceptual, logical, and physical. The first version of xLM (**ex**tensible **L**ogical **M**odel) was designed to support the logical model in that architecture. We refer the interested reader to one of the first reports we wrote on that work [9].

Recently, we picked up that work again as we were looking into the problem of managing and executing hybrid analytic flows. With some significant twists, we transformed the first xLM version into an encoding suitable for representing hybrid flows. Although little has been left from the original version, we kept the same name for historical reasons. In this document, we presented an overview of our current implementation for the *xLM* encoding.

Several people have contributed in one way or another to how xLM is shaped today and is described in this manuscript. The following list is most probably not exhaustive and should we omitted to mention someone, we sincerely apologize. Our special thanks[2] to HPL researchers: Umeshwar Dayal, Malu Castellanos, and Meichun Hsu; to our research visitor: Petar Jovanovic (UPC-BarcelonaTech); and to several HP ETL and BI practitioners, including Ravigopal Vennelakanti, Paul Watson, Paul Urban, John Bicknell, Rom Linhares, Heather Wainscott, Tom Harrocks, and Werner Rheeder.

[2]Several people have changed affiliations at the time this document was written; here, we list them with the affiliation they had when they contributed to the *xLM* project.

References

[1] P. Jovanovic, A. Simitsis, and K. Wilkinson. BabbleFlow: A translator for analytic data flow programs. In *ACM SIGMOD/PODS*, 2014.

[2] P. Jovanovic, A. Simitsis, and K. Wilkinson. Engine independence for logical analytic flows. In *IEEE ICDE*, 2014.

[3] C. P. Sayers, A. Simitsis, G. Koutrika, A. G. Gonzalez, D. T. Cantu, and M. Hsu. The Farm: where Pig scripts are bred and raised. In *ACM SIGMOD/PODS*, pages 1025–1028, 2013.

[4] R. C. Schank and L. G. Tesler. A conceptual parser for natural language. In *IJCAI*, pages 569–578, 1969.

[5] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *ACM SIGMOD/PODS*, pages 829–840, 2012.

[6] A. Simitsis, K. Wilkinson, and U. Dayal. Hybrid analytic flows - the case for optimization. *Fundam. Inform.*, 128(3):303–335, 2013.

[7] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu. HFMS: Managing the lifecycle and complexity of hybrid analytic data flows. In *IEEE ICDE*, pages 1174–1185, 2013.

[8] A. Simitsis, K. Wilkinson, and P. Jovanovic. xPAD: a platform for analytic data flows. In *ACM SIGMOD/PODS*, pages 1109–1112, 2013.

[9] K. Wilkinson, A. Simitsis, M. Castellanos, and U. Dayal. Leveraging business process models for ETL design. In *ER*, pages 15–30, 2010.

[10] E. S. K. Yu and J. Mylopoulos. From E-R to "A-R" - modelling strategic actor relationships for business process reengineering. In *ER*, pages 548–565, 1994.

Index