Engine Independence for Logical Analytic Flows

Petar Jovanovic #1◇, Alkis Simitsis *2, Kevin Wilkinson *3

Universitat Politecnica de Catalunya - BarcelonaTech, Spain
1 petar@essi.upc.edu

* HP Labs, Palo Alto, California, USA
2 alkis@hp.com, 3 kevin.wilkinson@hp.com

***Abstract*—A complex analytic flow in a modern enterprise may perform multiple, logically independent, tasks where each task uses a different processing engine. We term these multi-engine flows hybrid flows. Using multiple processing engines has advantages such as rapid deployment, better performance, lower cost, and so on. However, as the number and variety of these engines grows, developing and maintaining hybrid flows is a significant challenge because they are specified at a physical level and, so are hard to design and may break as the infrastructure evolves. We address this problem by enabling flow design at a logical level and automatic translation to physical flows. There are three main challenges. First, we describe how flows can be represented at a logical level, abstracting away details of any underlying processing engine. Second, we show how a physical flow, expressed in a programming language or some design GUI, can be imported and converted to a logical flow. In particular, we show how a hybrid flow comprising subflows in different languages can be imported and composed as a single, logical flow for subsequent manipulation. Third, we describe how a logical flow is translated into one or more physical flows for execution by the processing engines. The paper concludes with experimental results and example transformations that demonstrate the correctness and utility of our system.**

I. Introduction

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 the 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. This will only increase 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.

In this paper, we present a system that does this. We focus on three main challenges. First, we describe a language for encoding flows at a logical level. Second, we show how an existing, physical flow written for one processing engine, is imported and converted to a logical flow that is engine independent. Third, given a logical flow, we show how to generate a physical flow (and executable code) for a targeted processing engine. These physical to logical and logical to physical translations also support hybrid flows, i.e., flows that involve multiple engines. Our flow translators are components in a larger system, called Hybrid Flow Management System (HFMS), that includes modules for design, optimization, and execution of complex analytic flows [1]. Other tools, like ETL design GUIs, offer some separation between design and implementation, but the design is tool-specific. Our work goes beyond that. HFMS logical flows span engines and, most importantly, the engines are peers enabling data and function shipping between all.

The logical, engine-independent flow gives a unified, end-to-end view of the entire analytic computation. From this logical view, there is a number of possible and practical flow transformations. These may alter the flow design, but not its semantics (functionality). HFMS allows a flow processor to manipulate a logical flow between the physical to logical and logical to physical translations. Optimizing the logical flow is one possible transformation [2]. Or, one might decompose a single, large, complex flow into smaller subflows to reduce contention in a workload or to improve maintainability of the flow [1]. Conversely, one might compose a series of individual, connected flows into one large flow to improve performance. Or, a flow processor might generate documents about the flow, either as pseudo-code or as a natural language description [3].

In this paper, our focus is not on specific flow processors. Rather, our point is that a logical view of a flow simplifies and enhances flow processors. In fact, flow translation can be useful by itself. A not unusual scenario is to have an algorithm encoded for one engine (e.g., written in Map-Reduce) that you wish to apply to data in a different engine (e.g., database). Rather than ship the data to the algorithm, our flow translators enables shipping the algorithm to the data. Hence, our core contribution is that, by showing how to transform physical flows to logical flows and back, we enable new computations on hybrid flows that would otherwise be difficult to program over the original physical flows.

◇Work done while with HP Labs, Palo Alto.

The next section formalizes the translation problem. Section III presents an overview of our system, including encoding of flows and dictionaries for mapping between logical and physical elements. Section IV describes the physical to logical translation process and Section VI describes logical to physical. Section VII presents an evaluation of our system through use cases of flows running over three processing engines. The final sections discuss related work and conclusions.

II. Problem Formalization

A. Preliminaries

We represent 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}, and the edges E_Γ model the data flow among the vertices. A special class of operators includes the connectors, $V_{cn} \subset V_{op}$, which are discussed in Section IV. 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 $I_K=\{G_i\}$, $1 \le i \le K$ (K being the size of the set), each one having vertices assigned to a single engine. Each of these subflows is 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 $1 < j \le |V_\Gamma|$. Depending on the structure of Γ, we have three types of flows.

Definition 1. *A analytic flow is a multi-flow iff* $|I_K| > 1$.

Definition 2. *A multi-flow is a hybrid flow iff* $|\Phi_\Gamma| > 1$.

Each of the K subflows of a multi-flow is called a single flow. An analytic flow itself may be a single flow too.

Definition 3. *A analytic flow is a single flow iff* $|I_K| = 1$.

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

B. Logical and physical flows

A *logical* flow is independent of an execution engine. The 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 designed for a specific execution engine; e.g., specific RDBMS, specific map-reduce engine, specific ETL engine. The graph G_P representing such a physical flow contains the information needed to bound an operator for example to a specific implementation and engine.

C. Normalized flow

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 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 Section III, we present an implementation of L_N, called xLM. 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.

Definition 4. *We call engine agnostic xLM, denoted as a-xLM, the xLM encoding of a logical flow* G_L.

Definition 5. *We call engine specific xLM, denoted as s-xLM, the xLM encoding of a physical flow* G_P.

D. Dictionary

To enable conversions from one language to another and from physical to logical flows and vice versa, we keep a *dictionary* of mappings, D_M, between logical constructs and their valid physical incarnations in the supported languages. Example logical constructs are operators, functions, expressions, and data types. We denote as C_{L_i}, a construct expressed in a language L_i; e.g, for a logical construct we write C_{L_N}. We have two types of D_M mappings: (a) D_{MS}: (C_{L_i},C_{L_N}), these are, in general, one-to-one mappings used in physical to logical conversion, in order to identify the logical counterpart in L_N of a physical construct in L_i; and (b) D_{MO}: (C_{L_N},$\{C_{L_i}\}$), these are, in general, one-to-many mappings that map a logical construct to different physical implementations. (In Section IV-A, we discuss more complicated cases where more than one physical operator map to a single logical operator.)

In Section III, we describe an implementation of D_M and Figure 5 shows example dictionary entries.

E. Conversion process

For converting a physical flow to a logical one and vice versa, we are using a mapping system defined as follows.

Definition 6. *A mapping system is a triplet* ($\mathcal{S}$,$\mathcal{T}$,$\mathcal{M}_{\mathcal{S},\mathcal{T}}$), *where:* $\mathcal{S}$ *is a source graph,* $\mathcal{T}$ *is a target graph, and* $\mathcal{M}_{\mathcal{S},\mathcal{T}}$ *is a mapping between* $\mathcal{S}$ *and* $\mathcal{T}$.

Having a physical graph, G_P, of a flow, we can use a mapping $\mathcal{M}_{G_P,G_L}$ to convert it to a logical graph G_L, by applying the mapping onto all its vertices V_P. An implementation of $\mathcal{M}_{G_P,G_L}$ may be: $\mathcal{M}_{G_P,G_L}(optype, engine, impl)$ that can be used to probe the dictionary D_M for getting a logical operator corresponding to the physical implementation $impl$ of the physical operator $optype$ in the engine $engine$. Hence, starting from source code we can use $\mathcal{M}_{G_P,G_L}$ to produce an engine agnostic graph encoded in *a-xLM*.

A reverse mapping, $\mathcal{M}_{G_L,G_P}$, converts *a-xLM* to *s-xLM*. If a specific implementation of an operator is not defined in the mapping, then the system chooses the most efficient one according to the cost model used; if none exists, the system propagates the error to the user. Note that the *s-xLM* produced can be expressed in a language different from the one used in the original flow. We also define a mapping $\mathcal{M}_{G_P,G'_P}$ for changing the implementation of operations in a physical flow.

Finally, the *composition of mappings* is allowed. For example, the composite mapping:

$$\mathcal{M}_{G_{P_2},G'_{P_2}}(\mathcal{M}_{G_L,G_{P_2}}(\mathcal{M}_{G_{P_1},G_L}))$$

describes how a physical flow G_{P_1} expressed in a language L_1 can be first converted to a logical flow G_L, then to a physical flow G_{P_2} expressed in another language L_2, and then, to a different incarnation G'_{P_2} that uses alternative implementation for a subset of its operations on the same engine.

F. Problem statements

Problem 1 (Physical to Logical)**.** *Given an analytic flow* Γ*, we construct a logical flow* G_L *as follows:*

$$G_L = cmp(\bigcup_{1\leq i\leq K} \mathcal{M}_{G_{P_i},G_{L_i}}),\ \forall G_i \in I_K$$

That is, given an analytic flow Γ and a physical representation G_P of it, we produce a logical flow by converting first all single physical flows to logical flows, and then, we compose these flows to a unified, logical flow G_L. *Flow composition*, cmp, is described and solved in Section IV-B1. □

Problem 2 (Logical to Physical)**.** *Given a logical flow* G_L*, we construct an analytic flow* Γ' *as follows:*

$$\Gamma' = \bigcup_{1\leq j\leq K'} \mathcal{M}_{G_{L_j},G_{P_j}},\ \forall G_{L_j} \in dcmp(G_L)$$

That is, given a logical flow G_L corresponding to a flow Γ, we produce a semantically equivalent multi-flow Γ', by decomposing first the flow G_L to single, logical flows G_{L_j}, $1\leq j\leq K'$, each designed to run on a single engine, and then, we convert each G_{L_j} to a physical flow G_{P_j}. The poset of all G_{P_j} comprises Γ'. Notice that in general: (a) Φ'_Γ may differ from Φ_Γ, as Γ' may run on the same or a different set of engines than the original flow Γ; and (b) the number of single flows K in Γ and K' in Γ' may be different. *Flow decomposition*, $dcmp$, is described and solved in Section VI-A1. □

The two problems may be connected or not; i.e., starting from an analytic flow, the end goal might be to produce only its logical abstraction (Problem 1) or to produce another implementation of it (a combination of both Problems 1 and 2).

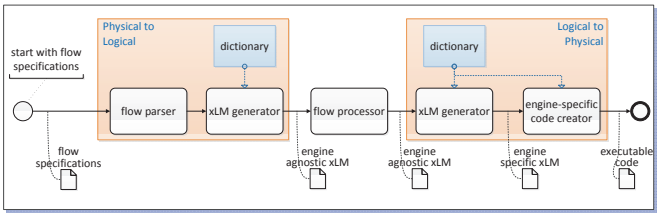


Fig. 1. Architecture of our solution

III. Architecture

This section describes our system architecture and implementations of our flow language, L_N, and the dictionary, D_M.

A. System overview

An overview of our approach is illustrated in Figure 1. We start with the flow specifications (e.g., a script in a programming language or metadata that encodes a physical flow) and we first convert it to a logical graph, encoded in engine agnostic xLM, *a-xLM*. This is the task of the 'physical to logical' module. There, we first parse the flow and then, we produce an xLM representation of the flow constructs using the dictionary. During parsing, it is possible, as we describe next, to collect statistics and cost estimates for the flow and its operations. Note also that the original flow may comprise more than one subflow (e.g., scripts) that may be written or be expressed in more than one programming language or forms. These are all translated into a single logical flow.

Next, a *flow processor* may transform the logical flow. Example processing modules are a flow optimizer (e.g., [2], [5], [6]), a collection statistics module (e.g., [2]), a flow execution scheduler (e.g., [1]), and so on. Detailing the different flow processors is out of the scope of this paper.

The 'logical to physical' module converts the engine agnostic, logical flow into an engine specific flow according to the engine selections made either by a flow processor or a flow administrator and using the xLM mappings stored in the dictionary. In some cases, the engine specific flow may be further processed by a flow processor; e.g., to apply engine specific optimizations to the physical flow (not shown in Figure 1). Finally, an 'engine specific code generator' module translates the engine specific xLM to executable code that can be dispatched to the processing engines.

B. Example

Figure 2 depicts a simple example of flow translation. In this scenario, we start with a PigLatin script, parse it and with the help of the dictionary, we translate the PigLatin operators to engine agnostic operators expressed in xLM (a-XLM). Then, a flow processor may change the flow. Finally, we produce engine specific, SQL-specific here, xLM (s-xLM) and from there, we generate a SQL incarnation of the original script. The figure highlights the process for an example filter operation. We discuss this process in more detail in the rest of the paper.

C. Flow encoding

HFMS uses xLM to serve as its language L_N [7]. xLM is a flow metadata language expressed, in its current implementation, in XML. It captures *structural information* of

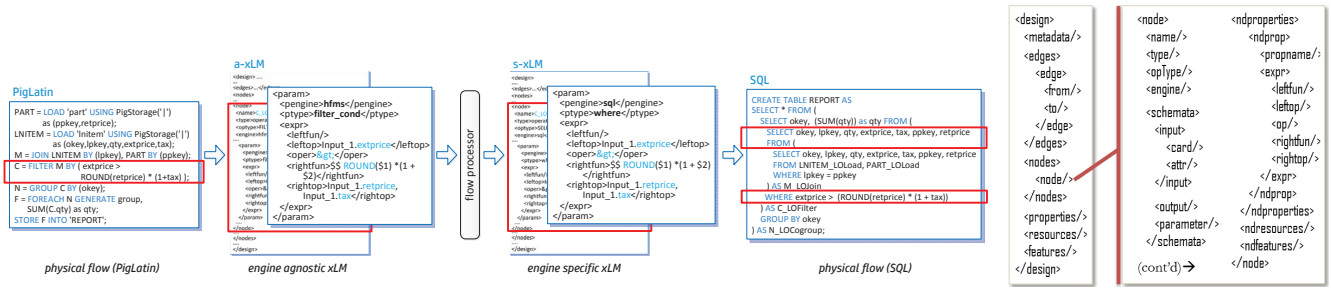


Fig. 2. Example process for translating a PigLatin script to SQL

Fig. 3. xLM elements

a flow, *design metadata* (e.g., functional and non-functional requirements, physical characteristics like resource allocation, positional information), *operator properties* (e.g., type, schemata, statistics, parameters and expressions needed for instantiating an operator, engine and implementation details, physical characteristics like memory budget), and so on.

xLM encodes a DAG and supports a rich set of operators, like relational algebra, analytic, machine learning or ETL-like operators. For the moment, however, our flow processors do not address fixpoints or iterative computations over a set of operators. In addition, HFMS treats operators with unknown or incomplete semantics as black-boxes. Processing flows containing black-box operators might not be optimal (e.g., optimization actions would be rather conservative), but it would be at minimum correct, respecting the semantics of the data flow and the schemata involved.

The two main xLM structural components are *design* and *node*. Figure 3 shows a skeleton of design (left) and node (right). *Design* describes a flow as a graph with its vertices and edges. It also describes flow properties, resources used, and features (e.g., location coordinates of the GUI elements) captured as expressions; e.g., we express 'timeWindow=2h' and 'max(failures)=3' as:

'timeWindow = 2h':
```
<leftop>timeWindow</leftop>
<oper>=</oper>
<rightop>2h</rightop>
```

'max(failures) = 3':
```
<leftfun>max</leftfun>
<leftop>failures</leftop>
<oper>=</oper>
<rightop>3</rightop>
```

Node describes a flow vertex with its name, type, operational type, engine, implementation, schemata, properties, resources, and features. The vertex type denotes if it is an operator or a data store. The operational type ($<opType>$) denotes the functionality of the operator; e.g., aggregator, tokenizer, sentiment miner. Flow vertices have input, output, and parameter schemata. Each schema represents a set of fields ($<attr>$) that have name, type, and properties (e.g., format, unit). Additional elements captured are properties like selectivity, throughput, path location; resources like i/o cost, allocated memory; and features like design coordinates.

xLM encodes both the logical and physical flows. Engine/language specific constructs can be nested in the corresponding element. For example, the xLM node for the filter of Figure 2 may contain multiple entries for its filter condition as in Figure 4. Based on the engine chosen for this filter, during flow conversion we may use the engine agnostic snippet ($pengine{=}hfms$, we use 'hfms' as a label of logical

```
<node>                                  (cnt'd)
<type>filter</type>                     <param>
...                                        <pengine>sql</pengine>
<param>                                    <ptype>where</ptype>
   <pengine>hfms</pengine>                 <expr>...</expr>
   <ptype>filter_cond</ptype>           </param>
   <expr>...</expr>                     <...>
</param>                                </node>
```

Fig. 4. Multiple language representations of a node in xLM

constructs) or the engine specific snippet ($pengine{=}sql$, here, 'sql' stands for generic SQL code, but it is also possible to specify a specific database engine) to produce the appropriate flow semantics. Beside expressions, the same logic is also followed to encode other engine specific elements like schemata attribute properties (e.g., data types) in different languages.

D. 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 implemented 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 (see Figure 5-left).

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.

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 implement the D_{MS} mappings and are engine specific, as they capture the intrinsic characteristics of an engine and map them to *a-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. There is also a variety

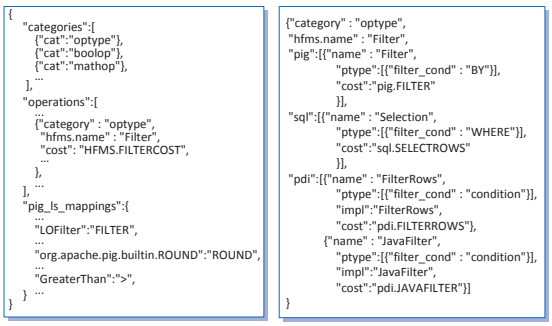


Fig. 5. Example entries in the dictionary

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 implement the D_{MO} mappings and describe the logical operations supported; e.g., operator and data store types. Figure 5-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 attributes, including: a logical operator name (e.g., `'hfms.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 5-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, etc.

Implementation. The dictionary can be implemented in various ways. In our implementation, we use a single file in JSON format; but other formats are straightforward to use.

Maintenance. Modifying the dictionary is a semi-automated process. It is extensible to new engines and implementations. For adding a new language or modifying an existing one, we use a template dictionary instance. When we finish entering the details for the new/updated language, then we use an automated mechanism for updating the dictionary accordingly.

E. Error handling

Translation failures may occur for various reasons; e.g., unavailable mappings or non-supported operations in an engine, connection or machine failures, runtime errors especially as we probe an engine to get an explain plan or run a sample flow to get runtime statistics. To the extent possible, we catch these errors and propagate them back to the user.

Errors such as incorrect dictionary mappings are harder to catch. Akin to proving a compiler is correct, it is hard to formally prove correctness properties. However, we can provide the user with a crude test-and-learn mechanism [8]

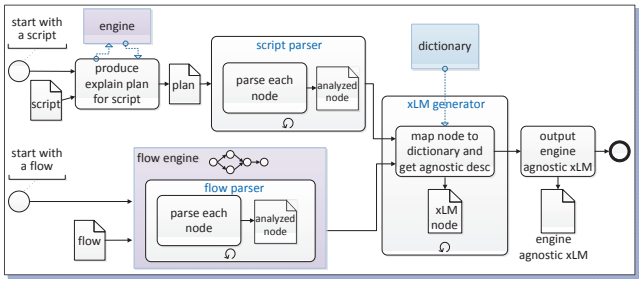


Fig. 6. Import single flow

to validate a mapping through experimentation with example data. In Section VII-B, we discuss experiments on correctness.

IV. Physical to Logical

This section deals with Problem 1 and describes how we convert a physical, analytic flow into an engine agnostic, logical flow. We distinguish two cases: single flows and multi-flows. Hybrid flows come as a variation of multi-flows.

A. Single flow

Converting a physical, single flow G_P to a logical flow G_L is described by a mapping $\mathcal{M}_{G_P,G_L}$. The main challenge here is to understand the semantics of a physical flow or else, of a script of execution code, and map it to a logical representation. For solving this, we first present how we parse execution code and convert it to xLM (see also Figure 6). Another challenge is how we deal with many-to-one mappings as in some languages a computation may require a different number of operators than in others. We can solve this either explicitly using the dictionary or implicitly as we discuss shortly.

One approach for parsing flow specifications would be to use a language specific parser to parse its source code. For some flows, like Map-Reduce programs, STORM programs, R scripts, etc., this might be the only solution. For several languages, e.g., SQL, there exist third-party parsers. In general, writing a language specific parser can be cumbersome. However, if the system provides an explain plan[1], as do database engines or Hadoop languages like PigLatin, we parse the plan instead of the code. We prefer using the explain plan because it is easier to parse and, coming from the engine, we know that it is syntactically correct. In addition, it provides extra information, which is not easy or sometimes even impossible to find by parsing the source code directly, such as cost estimates per operator, input data sizes, operator implementation, etc. Our script parser analyzes the *explain plan* (or the source code if the plan is not available) and gets engine specific information for each operator or data store of the flow. The script parser is engine specific and is added to the system as a plug-in.

The script parser pipelines information for each operator or data store to the xLM generator. This generator probes the D_M dictionary to map physical to logical operators. According to the mappings found, the engine specific information is replaced accordingly. For example: the engine specific operator type is transformed to a logical operator; an engine

[1] An explain plan shows the physical operators and data flow an engine uses to execute a flow.

specific expression is transformed to the form supported by the dictionary; the cost or data size estimates are used to feed the appropriate, implementation specific cost models that are essential for flow processing (e.g., optimization); and so on. Sometimes there are one-to-one mappings from one engine to another or to xLM. However, there are cases where the mappings are more complicated. For example, in PigLatin we can use two operators to specify aggregation (see Figure 2):

```
N = GROUP C BY (okey);
F = FOREACH N GENERATE group, SUM(C.qty) as qty;
```

Other programming languages perform the same calculation with a single operator. For example, in SQL we write:

```
SELECT SUM(qty) as qty FROM lnitm GROUP BY okey;
```

Many-to-one mappings can be resolved explicitly with a specific dictionary mapping or implicitly. For example, there are one-to-one mappings in the dictionary that map PigLatin.GROUPBY and PigLatin.FOREACH_GENERATE to logical GROUPER and PROJECT operators and from there to SQL.GROUPBY and SQL.PROJECT, respectively. Our language specific parser (i.e., the 'xLM flow processor' module in Figure 9) is enriched with additional smarts: when a logical GROUPER operator is followed by a logical PROJECT operator and iff the involved schemata match (e.g., PROJECT only uses the grouping attributes and the aggregates from the GROUPER) and the combination is valid, then we combine the two operators into one, generalized GROUPER. For no valid mappings, the process halts and asks for directions.

When all nodes have been processed, we output the engine agnostic *a-xLM*. This represents a logical flow abstraction.

Parsing the flow specification as described above is one way to create a logical flow. We use an alternative approach when an analytic flow has been created by a flow design tool like an ETL or a workflow tool. Instead of a script parser, it involves the engine itself (see left, bottom corner of Figure 6). This is more effective when access to the engine codebase is permitted. For example, we implemented this method in PDI, where we over-write the default printer method so that, for every flow node, we get the respective information (e.g., logical and physical) and pipeline it to the xLM generator [5]. The subsequent steps are as before.

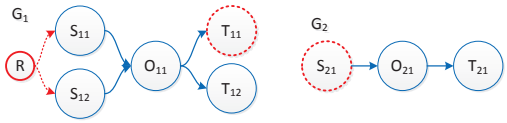
B. Multi-flow import

If the input flow comprises a multiplicity of single flows, possibly written in different languages, we work as follows. To abstract away the intrinsic characteristics of each language, we first process each single flow separately to create their logical counterparts. Then, we deal with three problems: (a) identify appropriate connect points that link single flows within a multi-flow, (b) choose appropriate connector operators, and (c) compose them into a single flow. Formally, this process is defined as a composition, cmp, over the physical to logical mappings: $cmp(\bigcup_{i\in I_K} \mathcal{M}_{G_{P_i},G_{L_i}})$.

1) Composition: To deal with these problems, we formulate flow composition as a series-parallel graph (SPG) composition problem [9]. SPGs are created by composing two-terminal graphs (TTG). We may have parallel or series composition of two TTGs. The former produces a new TTG created from the disjoint union of the two graphs by merging their sources and their sinks to create a new source and sink, respectively, for the new TTG. The latter produces a TTG created from the disjoint union of the two graphs by merging the sink of the first graph with the source of the second.

Formally, a TTG is a triple (G, s, t), where $G{=}(V, E)$ is an acyclic digraph and $s{,}t{\in}V$; s is called a source and t a sink of G. Assume two TTGs $((V_1, E_1), s_1, t_1)$ and $((V_2, E_2), s_2, t_2)$. Assuming $t_1{=}s_2$ (reads t_1 connects to s_2), the connect point is defined as $V_1{\cap}V_2{=}\{t_1\}$. Then, the series composition of the two TTGs is the TTG: $((V_1{\cup}V_2, E_1{\cup}E_2), s_1, t_2)$. Assuming $t_1{=}t_2$ and $s_1{=}s_2$ as connect points (i.e., $V_1{\cap}V_2{=}\{s_1, t_1\}$), the parallel composition of the two TTGs is the TTG: $((V_1{\cup}V_2, E_1{\cup}E_2), s_1, t_1)$.

In our case, a graph may have more than one source and/or sink vertices. However, it can be seen as a TTG if we work as follows. Considering a series of flows in a partial order, we begin with the first and last flows. If the first flow has more than one source, we add a dummy root vertex and connect the flow sources to the root. This dummy vertex has no semantics and used only for the composition. We do the same for the last flow if it has more than one sink vertex. (We work similarly if the partial order defines more than one 'first' or 'last' flows.) For all the other flows in the series, we define a single connect point between two connecting flows at a time. Hence, any flow in the series can now be seen as a TTG.



The above drawing shows an example involving two graphs G_1 and G_2. G_1 has two source nodes and by adding a dummy root, R, we create a single point of entry. For connecting G_1 to G_2 we need to identify a connect point. Assuming this is defined between vertices T_{11} and S_{21} –e.g., these two may represent the same physical storage, like a file– we are able to identify the TTGs involved in this composition as the graphs $R{\rightarrow}\ldots{\rightarrow}T_{11}$ and $S_{21}{\rightarrow}\ldots{\rightarrow}T_{21}$.

We identify connect points in two ways. First, these can be explicitly defined by the user, e.g., in the form of metadata. Alternatively, these points can be inferred based on an analysis of the input flows. When the flows contain connectors (described shortly), these may connect flows to the same data storage, which is then automatically used as a connect point; e.g., one flow writes to a file or a table and a following flow reads from it. We can also discover *compatible* vertices between two flows, based on the similarity and compatibility of their output (for vertices in the first flow) and input (for vertices in the second flow) schemata. We process pairs of flows based on their execution order[2] and search for possible matchings between the sink vertices of the preceding flow and the source

[2]Single flows may come as different scripts with metadata determining their execution order. Without such a metadata, the user defines the order. Here, without loss of generality we assume that the execution order is provided.

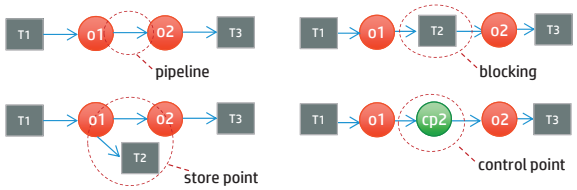


Fig. 7. Connection types

vertices of the following flow. Exact matches are candidate connect points. For approximate matches, we perform conservative schema refactoring by changing the names of the inputs and/or outputs. An application of this technique has been used for demonstrating composition of PigLatin scripts [3]. User feedback is requested if no match is found. Advanced schema matching and mapping techniques [10], [11] could be used too, but we consider this is as an interesting future direction.

Note that flow composition is not shown in Figure 6, which only shows import for single flows. For multi-flows, a 'flow composer' module analyzes the individual, logical graphs produced and performs the composition as discussed.

2) Connectors: As we identify candidate connect points, we also determine appropriate operators to realize the connections. At a logical level, a link between single flows is modeled in the flow with a *connector* operator. At a physical level, connectors delimit subflow boundaries and, possibly, *engine barriers*, i.e., operators on either side of a connector may run on different engines. Based on the connection semantics, a number of connection types are supported: pipeline, blocking, check point, and control point (Figure 7). Check points can be useful for recoverability or synchronization. Control points moderate the data flow according to a condition; e.g., wait until $x\%$ of the data has arrived, send batches every y time units, etc. As a syntactic sugar, in the first three cases, an explicit connector operator may be omitted from the flow graph design.

As said, given a multi-flow, the connectors may be specified explicitly, through metadata associated with the flow, or determined implicitly. Implicit connectors are located during the physical to logical conversion by looking for operator patterns. Figure 8 illustrates two example types of subflow connectivity.

Figure 8-left shows an inter-engine connection from a PigLatin subflow to a SQL subflow. The connector executes on the consumer subflow and reads data from an HDFS file (many db providers offer such connectors). In such cases, the connect point between the two subflows can be inferred by looking into their sink and source nodes as discussed.

Figure 8-right shows two subflows where no connectors have been specified nor can any be inferred. In this case, we analyze the terminal nodes of the graphs and if we identify compatible vertices, we add an appropriate connector. If no match is found, then we ask the user to resolve the ambiguity. Suppose the user specifies a connector between HDFS $file1$ ($F2$) and database table $T1$. The figure shows two possible connectors. We can automatically enrich the producer subflow with a connector at its output, which will eventually be converted to PigLatin during code generation –see Section VI; this is shown in bold, red in the (I) solution. Alternatively, we can add extra operators packaged as glue code to read from the storage

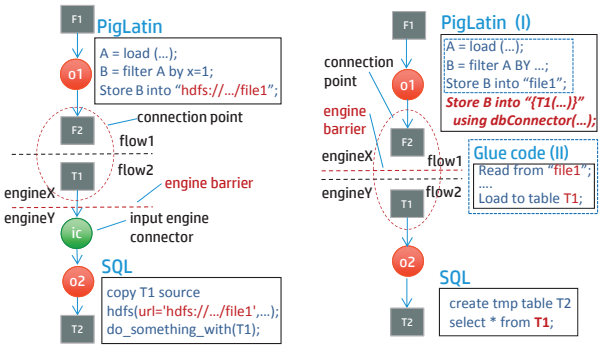


Fig. 8. Example connectors for hybrid flows

point of the first flow, $file1$, and load data to the matching, entry point of the second flow, $T1$; the (II) solution includes the parts of the scripts encircled by the blue dotted line. The choice between the two solutions depends on parameters like availability (not all engines support explicit connectors), costs, or user choice. Regarding costs of different connecting paths, we use a cost model based on a combination of micro-benchmarks and runtime statistics monitoring the system status (e.g., disk i/o, network congestion). A discussion on the costs is outside the scope of this paper (more details in [2]).

To uniformly capture the different cases in $\mathcal{M}_{G_P,G_L}$, metadata for logical connectors includes semantics of these cases. A flow processor may exploit it as needed. This metadata is also used in converting a logical connector to a physical connector. The metadata is captured as xLM properties like: pipeline or blocking connections; disk or memory storage; the type of the encapsulated data stores for blocking connectors; the flow lineage, in which single flow a part of the multi-flow originally belongs to; file paths; db connections; partitioning and clustering schema; etc. As an aside, in some cases it is reasonable to convert a blocking connector (e.g., a storage object) to a pipelined connector. For example, if the storage object is a temporary table in a SQL script and it only has one reader, then the flow that produces the data can be pipeline connected to the flow that consumes the table. But we need to be careful, since there may be other objects (e.g., other flows) that may use this storage.

V. Flow Processor

A flow processor takes as input a logical flow graph, performs some transformation on that graph, and produces as output a second logical graph that is functionally equivalent to the input but with different properties.

As discussed, the HFMS optimizer is one example of a flow processor. Another flow processor might be used to decompose a long flow into subflows or to compose multiple subflows into a single flow. A more detailed analysis of the modules that handle flow processing can be found elsewhere [1].

In the context of this paper, an important function of a flow processor is to alter connectors. For example, flow composition would remove connectors. Flow decomposition would add connectors. Flow optimization may include function shipping, i.e., moving an operator from one engine to another, which

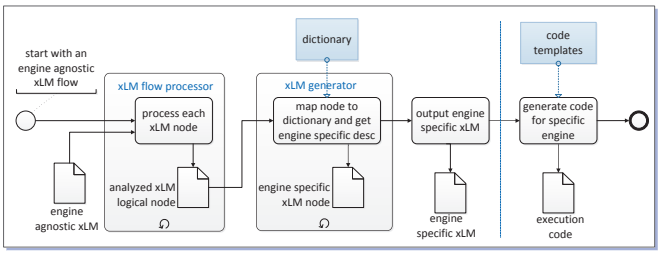


Fig. 9. Convert *a-xLM* to *s-xLM* and generate code

involves swapping the position of an operator and a connector. Retargeting a single flow from one engine to another involves modifying the metadata for the connector to the new engine. Once the connectors are determined, so are the engine boundaries. Since connectors link subflows on different engines, given the engine assignment, a physical flow can be generated. Note that the engine assignment can also be defined manually by a system administrator.

VI. Logical to Physical

This section deals with Problem 2 and describes how we convert an engine agnostic, logical flow first into a physical, engine specific flow, and then, to executable code that can be dispatched to execution engines.

A. Creating an engine specific flow

A logical graph is engine agnostic, *a-xLM*. A flow processor determines on what engine(s) it will be executed and thus, we convert *a-xLM* to *s-xLM* using this information. If the entire flow has been assigned to a single engine, we proceed directly to the conversion. If the operators of the logical flow have been assigned to multiple engines, then before converting G_L to G_P, we decompose the logical flow, $dcmp(G_L)$, to single flows, G_{L_j}, each running on one engine (see Problem 2).

1) Decomposition: For a logical flow $G_L{=}(V_L, E_L)$, the connectors between subflows assigned to different engines are engine barriers (see also Figure 8). For creating single flows, we split the logical graph at the engine barriers. The result of this split is a set of weak graph components, each one corresponding to a single flow. A weak component is the maximal subgraph in which all pairs of vertices in the subgraph are reachable from one another in the underlying subgraph. The weak components are stored in a list according to a topological sort of the original flow; each weak component, i.e., single flow, is prioritized according to the topological order of its vertices in the original, composed flow. These priorities will determine the execution order of the scripts produced in code generation, as we describe shortly.

2) Conversion: Converting a logical flow G_{L_j} to a physical flow G_{P_j} is described by a mapping $\mathcal{M}_{G_{L_j},G_{P_j}}$ (see also Figure 9). G_{L_j} is encoded in *a-xLM*, which describes the main flow structure, but also carries engine specific details captured during $\mathcal{M}_{G_P,G_L}$, like paths to data storage or design metadata. These details are now used for producing *s-xLM*. For space considerations, we omit an extensive description of the conversion of *a-xLM* constructs into *s-xLM*. The dictionary is a centerpiece in this process as it contains templates with default

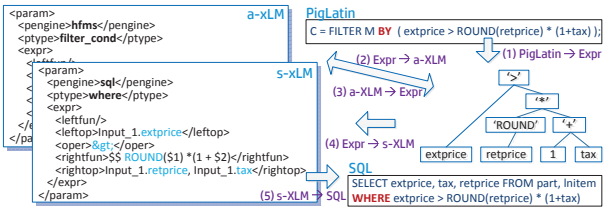


Fig. 10. Example expression tree usage

metadata information for operator representation in different engines as part of the language specific mappings, D_{MS}.

We elaborate on the conversion of *expressions* as an indicative example. Operators have parameter schemata describing their functionality. Typically, this information is represented with an expression. Expression formats vary across languages so, during the import phase, the original expressions are transformed into the logical format that our system understands. Now, we perform the reverse operation and convert them to a form that the targeted engine can process. There are two issues.

First, we determine the context of the expression. For example, an expression may describe a filter condition or may define a new projection schema; e.g., in SQL these correspond to conditions in the WHERE and SELECT clauses, respectively.

Second, apart from the metadata, we need to parse the expression itself. Expressions, like mathematic expressions or built-in functions, may be represented differently in different engines. For example, a conjunction may be written as 'X AND Y' or 'AND(X,Y)' or 'X && Y', etc. Similarly, built-in functions may also differ from engine to engine. For example, SQRT(a) in SQL, Math.sqrt(a) in PDI, org.apache.pig.builtin.SQRT(a) in PigLatin, etc. Hence, during the conversion from physical to logical, we map the engine specific forms to logical forms as follows. Starting from an engine specific expression, we first build an expression tree, whose nodes are operators and built-in functions found in the expression, while the leaves of the tree are the attributes and constants included in the expression. Now, we reuse this tree for producing an engine specific expression (possibly for another engine). We use the dictionary to retrieve suitable mappings for the constructs of the expression (e.g., AND or &&, built-in functions) and the proper usage of such expressions (e.g., 'X AND Y' or 'AND(X,Y)').

Figure 10 shows a transformation for the FILTER of Figure 2. As discussed in Section VI, starting from PigLatin, we identify the expression describing its filter condition (step 1 in Figure 10), create an expression tree, and produce *a-xLM* (step 2). We now reuse the expression tree (step 3) for creating *s-xLM* for SQL (step 4), and finally, produce SQL code (step 5).

An overview of the conversion of *a-xLM* to *s-xLM* is depicted in Figure 9 (on the left of the vertical dashed line).

B. Code generation

This section describes how we generate code from *s-xLM*; see also Figure 9, on the right of the vertical dashed line.

We generate code separately for each of the decomposed, single engine specific flows; these tasks are done in parallel. In each single flow, we process the flow operations following a topological sort of the flow graph to ensure that before gener-

ating a code snippet all its prerequisites have been created. At the end, we produce orchestration code to package (e.g., as a shell script) the code pieces for execution.

Next, we describe a template mechanism that enables code generation and then, we give an example of code generation for SQL –other languages are omitted for space considerations.

1) Templates: For each operator implementation in the dictionary, we define a code template with formal parameters, which can be used to produce code for a graph construct. We distinguish two groups of templates: *code* and *metadata* templates. Code templates produce executable code directly when they are appropriately instantiated (e.g., a template for producing a SQL or PigLatin statement). Metadata templates produce meta-code interpretable by design tools, like ETL tools, which store the flow metadata in a specific format (e.g., in an XML or JSON script). This format is later used for importing a flow into those external tools, and then at compile time, the tools create the appropriate execution code.

In our current implementation, we support a fairly large number of operators (more than a hundred) and it is relatively easy to register a new one. Our code templates are expressed in JavaScript and are invoked by a script engine at runtime. For the example FILTER of Figure 5, simplified example code templates for PigLatin and SQL are as follows:

```
function pig_FILTER(args) {
  return args.operator + " =\n  FILTER "
        + args.producer + "\n BY " + args.param + ";";
}
function sql_SELECTROWS(args) {
  w=(args.where=="")?"":" \nWHERE " + args.where;
  g=(args.groupby=="")?"":" \nGROUP BY " + args.groupby;
  h=(args.having=="")?"":" \nHAVING " + args.having;
  o=(args.orderby=="")?"":" \nORDER BY " + args.orderby;
  return "SELECT "+args.out+"\nFROM "+args.in+w+g+h+o+";";
}
```

When code is generated, the formal parameters are replaced by the actual parameters from the engine specific flow in order to generate a code snippet for the operator. The operator metadata also includes formal schemata for the input and output data sets, which specify the required and optional fields of each input record and the fields of the output records. In addition, there is a parameter schema that specifies the formal parameters, if any, needed for the operator. The data stores also have associated metadata, like a size estimate of the data set, as number of records, and a schema for the records. For data stores, the metadata also includes the location of the data store.

The code templates for the above, example filter operator can be instantiated as `Z = FILTER X BY Y` (PigLatin) and `SELECT Z' FROM X WHERE Y` (SQL), assuming that the filter operator has a parameter schema `Y`, a producer operation `X`, and a consumer operation `Z` with input schema `Z'`.

In addition to generating code for operators, we can also generate code that implements data flow between operators. There are two cases. If both producer and consumer operators use the same underlying execution engine, the execution form for that engine may have syntax for connecting the two operators, e.g., nested SQL clauses or pipelining Unix shell scripts. If no syntax is available, e.g., PigLatin, the output of the producer can be placed in a temporary data set which is then specified as an input to the consumer. If either the producer or the consumer is a connector, then, as we discussed in Section IV-B, connector code snippets are generated to transfer the dataset across engine.

2) Code generation - The SQL case: For space considerations, we describe code generation with *code templates*. We work similarly with metadata templates.

The designer may influence the style of generated code. For example, a nesting flow may run faster. But a decomposed flow splits the computation into shorter phases that use fewer resources and under certain conditions may reduce contention for system resources. Also, a designer might want to decompose a complicated, nested SQL query (e.g., perhaps a query generated by a reporting tool) into a script of sub-flows in order to resolve a data consistency issue (i.e., by exposing intermediate results). The final form of the executable depends on the designer's objectives. There are trade-offs among response time, throughput, maintainability, etc. in using these forms. In different ways and at different times, both styles can be useful.

HFMS offers an interactive environment, where a designer may explore and test alternatives for flow execution by indicating the degree of pipelining desired for the executable code. At one end of the spectrum, blocking may be specified. In this case, every operator is made into a blocking operator by simply inserting the output of the operator into a temporary table. At the other end of the spectrum, pipelining may be specified. In this case, the entire flow is generated –if possible– as a single SQL statement by using nested SQL. Note that this does not convert a blocking operator into non-blocking. For example, even sort operators are nested even though this operator is blocking. However, it enables the SQL engine to optimize the entire flow as a single statement which should produce more efficient code, but perhaps less readable or maintainable.

Between the two extremes, a user may specify pipelining that uses temporarily tables either more or less aggressively. We use heuristics to determine where to insert the temporary tables, like after a naturally occurring blocking operator, since no pipelining can occur with such an operator anyway. We also use estimates of the size of the datasets as well as available resources to determine whether to use temporary tables or not. For example a small data set may easily be buffered in memory to reduce the amount of I/O. A large data set may need to be spooled to disk even if pipelining is specified.

Under the hood, the user or a flow processor may choose a nesting strategy. We first split the flow graph at breaking points into query paths. Breaking points denote where a new subquery should start or a temporary table may be needed. Example breaking points are operators with multiple output (e.g., Router) and multiple input (e.g., Merger) paths. Then, we process each query path and construct a SQL subquery; depending on the nesting level, this may be a single query or it may contain temporary tables. If a query path contains a data storage (e.g., source table, intermediate table) then an appropriate DDL statement is constructed too.

For creating a nesting query, we use template placeholders in the SQL expression of each operator and afterwards, we

replace it with an appropriately constructed subquery. For example, the SQL expressions for the operators of a subflow T1→N2→N3 may be as follows:

```
T1: create table T1 ...; --T1 may already exist too
N2: select * from ##T1##;
N3: select * from ##N2##;
```

Following a topological sort, we resolve dependencies by replacing the placeholders ##T1## and ##N2## with the respective expressions. Based on the nesting level, we create the final query as follows. For maximum nesting, we have:

```
select * from (select * from T1) as N2;
```

For a lesser nesting level and assuming that N2 is a breaking point, the templates for N2 and N3 would be:

```
N2: create temp table TEMP_N2 as select * from ##T1##;
N3: select * from ##N2##;
```

and the final query is:

```
create temp table TEMP_N2 as select * from T1;
select * from TEMP_N2;
```

Finally, we produce the final SQL expressions for tables, temporary tables, and the main (nested) query. Due to space considerations, we omit a formal description of this algorithm.

VII. Evaluation

We demonstrate the utility of our techniques by showing representative results of our system behavior.

A. Preliminaries

Implementation. The system we described here comprises an API module for our hybrid flow management system (HFMS). The API is implemented in about 25K lines of Java code. The dictionary is implemented in JSON. xLM template operators are built with Apache Velocity 1.7. Code generation is performed with embedded JavaScript. For engine support, when we do not use code parsers, we parse execution plans that we get either by probing the engine (e.g., JDBC for SQL) or by using an external library of the language without connecting to the engine (e.g., PigServer for PigLatin). We also modified the open source codebase of PDI to make it xLM aware.

Methodology. In the evaluation, we focus on three languages: SQL (sql) running on a commercial DBMS, PigLatin (pig) running on Hadoop 0.20.2, and PDI flows (pdi) running on the community edition of Pentaho PDI 4.4.0. Our intention is to show that our techniques work for both declarative and procedural languages, but also for flows expressed as metadata.

We used flows written in all three languages implementing the logic of 15 TPC-H and 15 TPC-DS queries. We chose these as representative examples of transactional and analytical scenarios. We also used 10 custom made flows combining ETL and analytical logic and having their subflows implemented either in one (multi-flows) or in two (hybrid flows) languages.

B. Experiments

1) Correctness: We evaluated the correctness of our methods in two ways: (a) comparing the code generated, and (b) examining the results produced by executing all scenarios used in all three code variants: SQL, PigLatin, and PDI.

Compare code. We compared the original code and execution plans (where available) to both the code produced and the execution plans the engines created for the new code. For a fair comparison, (a) we created snippets in the language they were originally written in and disabled any interaction with the flow processors, and (b) we brought the original and new snippets in the same format: same capital/small letter format, trim extra spaces and indentation, etc. We also round-tripped the translation of scripts starting from one language, converting them to the other two languages in a random order, and then back to the original language. All scripts compared were semantically equivalent. However, there were also differences as follows.

Different variable names: this is due to the code re-factoring done in composition for enabling schema matches and due to the operator and variable name substitution performed to add semantics to a script; e.g., change one-char names in PigLatin like 'C=Join...' with 'C_LOJoin=Join...'.

Different placing order: operators belonging to the same execution order class –i.e., operators without dependencies amongst them– may be placed differently; e.g., two operators $filter_1$ and $filter_2$ may be reordered as $filter_2$ and $filter_1$.

Different flavor of code: the code produced may have different nesting degree (e.g., number of intermediate tables).

Compare results. We ran all flows in their original configuration. We then produced new flows by varying combinations of the following attributes: target engines (single and multiple engines) and decomposition level (ranging from full nested to full blocking). For all scripts, the produced results after running code variants for the same flow were identical to the original, except, for row order, when it was not specified in the flow.

As a side note, the results we report here are indicative of our effort to evaluate our translators. During implementation, we have gone through the language specification, translated each operator, and ran tests with it. Of course these tests are with specific input values and so, while great for catching obvious programming errors, they may miss data-dependent errors or subtle implementation differences between engines. In addition, we have not performed extensive tests to evaluate program equivalence in side effects or failure conditions.

2) Performance: We performed experiments focusing on the behavior of our system. All other features of HFMS (e.g., optimizer, workload manager) were disabled.

Physical to Logical. Figure 11(a) shows execution times for creating a logical graph starting from the 15 TPC-H flows written in sql and pig. For sql and pig the different phases during import are: parse a plan (pp) and create a logical graph (axlm). For pdi, since we changed its codebase to produce directly xLM, there is only one phase, axlm. For space considerations, we do not plot the results for pdi, but for the same flows they range between 5 and 20 seconds.

The flows in the figure are placed in increasing order of complexity. We consider complexity related to the flow size and type of operators performed; e.g., nesting or operators with more than one input or output schemata make parsing more complex. As seen, parsing gets slower with flow complexity.

Our parsers work differently. For sql and pig, some of the smarts in understanding the operations involved, nesting, etc. can be inferred from the plan. In pdi and when we parse code

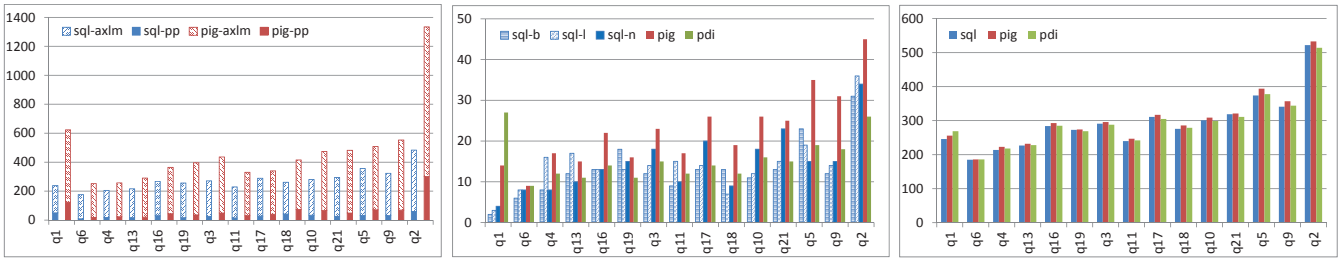


Fig. 11. Time analysis for 15 TPC-H flows (times in msec): (a) from SQL/Pig code to *a-xLM*, (b) from *s-xLM* to code, (c) from SQL code to xLM to code

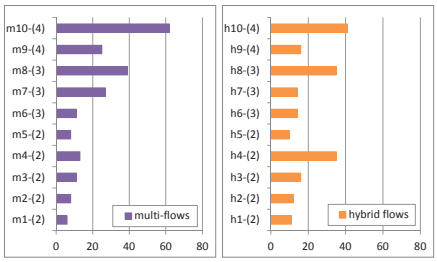


Fig. 12. Flow composition (times in msec)

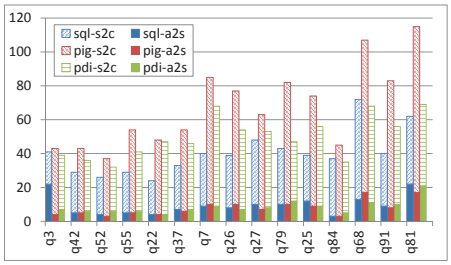


Fig. 13. From *a-xLM* to *s-xLM* to code (times in msec)

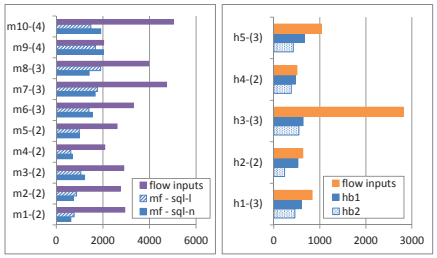


Fig. 14. Flow execution (times in msec)

directly, some cycles are also spent for identifying the flow semantics, get extra information like runtime statistics, sizes of the involved data sets, etc. In addition, especially for pdi, some extra time (an avg 2-3sec per flow) is spent to ensure correct transfer of metadata associated with the flow design, not the semantics. A faster implementation is possible by changing the internals of PDI engine, but since we wanted to leave the lightest possible footprint and not be highly dependent from future versions of the engine, we compromised with paying an extra cost for operations not directly related to *a-xLM* creation.

We performed similar experiments with the 15 TPC-DS flows. The times for converting them to *a-xLM* starting from sql and pig follow the same trends as with the TPC-H flows. axlm dominates the import (an average of 497msec), while pp was cheaper (avg 88msec). Import for pdi was again slower, between 7 and 24 seconds, for the reasons we discussed.

We performed composition of 10 multi-flows (m) and 10 hybrid flows (h) (in pig and sql). We omit the results for each subflow import, since the trend is as before. Figure 12 shows the results of composing the individual subflows into a single logical graph. The y-axis shows the id of each flow and in a parenthesis, the number of its subflows, K. In most cases, composition takes less than 40msec. Time difference for same K relates to the identification of connection points (e.g., the hybrid $h4$-(2) comes with no pre-specified connectors).

Logical to Physical. The avg time for converting *a-xLM* to *s-xLM*, for the TPC-H flows tested was 3msec and for producing code from *s-xLM* was 23 for pig, 15 for pdi, 14 for sql (all times in msec). Figure 11(b) shows time needed to parse *s-xLM* and generate code for all engines pig, pdi, and sql –in three variants, blocking sql-b, moderate use of intermediate storage sql-l, and nesting sql-n. Again, the flows in the figure are placed in increasing order of complexity. In general, we produce sql faster than the other two languages, but on average, this is close to pdi code creation. For pig, we spend some extra cycles to identify n-to-m mappings and decompose operators like the GROUPER operation discussed in Section IV-A.

Figure 11(c) shows end-to-end time needed to convert a flow written in sql to xLM and from there either to pig, pdi or sql again. These times are pretty similar, and as the figure shows too, our system needs less than 500msec for a full translation. There is similar behavior when we start from pig (on avg less than 500msec) or pdi, with the latter being slower (5 to 20 seconds) due to the import cost of PDI flows.

We observed similar trends for the TPC-DS flows. These are more complex flows, so the conversion *a-xLM* to *s-xLM* was a bit slower, averaging 10msec for all *s-xLM* variants. Figure 13 shows the end-to-end times for the three languages tested.

For the 10 hybrid flows, the time to decompose each one to graphs assigned to pig and sql is less than 2msec. After that, code generation for each subflow is as in the other cases.

Execution. The focus here is on flow translators, not flow processors. However, an indicative example of how translation enables advanced opportunities in flow execution is shown in Figure 14. We tested 10 multi-flows and 5 hybrid flows comprising a varying number n of single flows, denoted as m_i-(n) and h_i-(n), respectively. Multi-flows were improved after composing them as one flow, either as sql-l or sql-n, as the database optimizes the whole. hb1 shows that just composition can improve hybrid flows, because each engine optimizes better its subflow. hb2 shows the effect of additional optimization performed to the entire flow (e.g., shipping operations from sql to pig). For space constraints, we show here only the sql part of the hybrid; pig is much slower (avg 1min) but is also affected positively by hb2. These are examples. A detailed study on the benefits in running processes as hybrid flows and in modifying a hybrid flow workload at runtime can be found in [1], [6].

3) Discussion: As seen here, and from our overall user experience, the translation is fairly fast and can be done at run time. If used by a user, the translation is interactive and offers an 'instantaneous' feeling. If used by a flow processor, it does not add much to the latency of flow execution. This makes the flow translators a useful component of HFMS as it enables features like optimization (provides a single view of the end-to-end flow), workload management at a flow execution run time (enables new actions like *flow decomposition* and *flow*

shipping when the targeted engine is unavailable or overloaded), scheduling (flexibility in code rewriting allows more options in cloud resource provisioning and scheduling), etc.

On the other hand, our translators are useful as an autonomous system too. An example use case is code migration. It can be used to migrate legacy routines to a new language (or even a new version of the language) or to convert code running on one system to another; e.g., from one database engine to another that may use different SQL syntax.

VIII. Related Work

Various systems generate a single type of executable code from a logical graph. Some systems generate code within a particular family of targets; e.g., database query systems generate SQL for a variety of database engines. Other systems generate executable code for more than one execution engine like ETL systems. But, these systems are inflexible in that they generate identical executable code for a given graph. Our system generates various executable forms from a single graph.

Research on federated database systems considered query processing across multiple database engines (e.g., Garlic [12], Multibase [13], Pegasus [14]). However, our work extends to a large variety of execution engines. Some systems consider hybrid execution in which queries span two execution engines, namely a database and a Map-Reduce engines; e.g., HadoopDB/Hadapt [15], PolyBase [16], Teradata Aster [17]. These are closer to our hybrid flows, but they are not designed as a general framework over multiple engines. They follow an ego-centric logic providing connectors to other engines, whereas we consider engines as equal peers.

There is research work in code generation for integration flows. Orchid converts declarative mapping specification into data flow specifications; but the focus is on ETL jobs [18]. GCIP generates code for integration systems like ETL and EAI, but not for other execution engines [19]. There is also work on SQL generation that applies simple rules to compose SQL queries (without subqueries) by identifying insert and create/drop table statements and eliminating intermediate steps [20]. Another work translates BPEL/SQL to BPEL/SQL using process graph model, but the focus is on business processes [21]. We differ for both these works, since we can digest more complex queries and flows written in other languages too, and can convert them back to multiple languages.

IX. Conclusions

Our work is motivated by the emerging trend of using a diverse set of processing engines within a complex analytic flow. Implementing and maintaining these hybrid flows at a physical level is a burden and we propose using logical flows for engine independence. In this paper, we describe how engine independence is supported by HFMS, our Hybrid Flow Management System. We show how our language, xLM, can encode flows, both logical and physical elements. We then show how a physical flow is imported into HFMS and translated to a logical flow that is not bound to a specific engine. That logical flow can then be translated back to a physical flow and then to executable code for another processing engine. These translations are enabled through a dictionary and engine specific mappings that are easy to extend to new engines and new operators. To represent links between subflows of a larger flow, HFMS has a set of connector operators that capture the semantics of the connections between subflows.

An evaluation of the HFMS translators was demonstrated using three processing engines, Hadoop/PigLatin, Pentaho PDI, and a SQL engine. Evidence of correctness is shown by round-tripping, i.e., given a flow for engine$_x$, HFMS translates it to a logical flow and then to a physical flow for engine$_x$ or engine$_y$ that produces identical results. Examples of the utility were presented including composition and decomposition.

We acknowledge there are language constructs that will be difficult to translate. For example, the semantics of user-defined functions (UDFs) vary across engines, so not all can be translated (e.g., some support only scalar, some aggregation, some are multi-valued, etc.). However, we believe our approach is sufficiently general to model and translate a sufficiently large class of analytic flows. Some of our next steps include incorporating additional engines of interest and enhancing the automation of flow composition by using advanced schema matching and mapping techniques [10], [11].

References

[1] A. Simitsis et al., "HFMS: Managing the lifecycle and complexity of hybrid analytic data flows," in ICDE, 2013, pp. 1174–1185.

[2] A. Simitsis, K. Wilkinson, and U. Dayal, "Hybrid analytic flows - the case for optimization," Fundamenta Informaticae, vol. 128, no. 3, pp. 303–335, 2013.

[3] C. P. Sayers et al., "The Farm: where pig scripts are bred and raised," in SIGMOD, 2013, pp. 1025–1028.

[4] R. C. Schank and L. G. Tesler, "A conceptual parser for natural language," in IJCAI, 1969, pp. 569–578.

[5] A. Simitsis, K. Wilkinson, and P. Jovanovic, "xPAD: a platform for analytic data flows," in SIGMOD Conference, 2013, pp. 1109–1112.

[6] A. Simitsis et al., "Optimizing Analytic Data Flows for Multiple Execution Engines," in SIGMOD, 2012, pp. 829–840.

[7] A. Simitsis and K. Wilkinson, "The specification for xLM: an encoding for analytic flows," HP Labs, Tech. Rep., 2014.

[8] T. H. Davenport, "How to design smart business experiments," Harvard Business Review, pp. 68–76, 2009.

[9] R. Duffin, "Topology of series-parallel networks," J. of Mathematical Analysis and Applications, vol. 10, no. 2, pp. 303–318, 1965.

[10] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," VLDB J., vol. 10, no. 4, pp. 334–350, 2001.

[11] P. G. Kolaitis, "Schema mappings, data exchange, and metadata management," in PODS, 2005, pp. 61–75.

[12] M. T. Roth et al., "The garlic project," in SIGMOD, 1996, p. 557.

[13] U. Dayal, "Processing queries over generalization hierarchies in a multidatabase system," in VLDB, 1983, pp. 342–353.

[14] W. Du, R. Krishnamurthy, and M.-C. Shan, "Query optimization in a heterogeneous DBMS," in VLDB, 1992, pp. 277–291.

[15] A. Abouzeid et al., "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads," PVLDB, vol. 2, no. 1, 2009.

[16] D. J. DeWitt et al., "Split query processing in Polybase," in SIGMOD Conference, 2013, pp. 1255–1266.

[17] E. Friedman, P. M. Pawlowski, and J. Cieslewicz, "SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," PVLDB, vol. 2, no. 2, pp. 1402–1413, 2009.

[18] S. Dessloch et al., "Orchid: Integrating schema mapping and ETL," in ICDE, 2008, pp. 1307–1316.

[19] M. Böhm et al., "GCIP: exploiting the generation and optimization of integration processes," in EDBT, 2009, pp. 1128–1131.

[20] T. Kraft et al., "Coarse-grained optimization: Techniques for rewriting SQL statement sequences," in VLDB, 2003, pp. 488–499.

[21] M. Vrhovnik et al., "An approach to optimize data processing in business processes," in VLDB, 2007, pp. 615–626.