BabbleFlow - A Translator for Analytic Data Flow Programs

Petar Jovanovic*
UPC - BarcelonaTech
Barcelona, Spain
petar@essi.upc.edu

Alkis Simitsis
HP Labs
Palo Alto, California, USA
alkis@hp.com

Kevin Wilkinson
HP Labs
Palo Alto, California, USA
kevin.wilkinson@hp.com

ABSTRACT

A complex analytic data flow may perform multiple, inter-dependent tasks where each task uses a different processing engine. Such a multi-engine flow, termed a hybrid flow, may comprise subflows written in more than one programming language. However, as the number and variety of these engines grow, developing and maintaining hybrid flows at the physical level becomes increasingly challenging. To address this problem, we present BabbleFlow, a system for enabling flow design at a logical level and automatic translation to physical flows. BabbleFlow translates a hybrid flow expressed in a number of languages to a semantically equivalent hybrid flow expressed in the same or a different set of languages. To this end, it composes the multiple physical flows of a hybrid flow into a single logical representation expressed in a unified flow language called xLM. In doing so, it enables a number of graph transformations such as (de-)composition and optimization. Then, it converts the, possibly transformed, xLM data flow graph into an executable form by expressing it in one or more target programming languages.

Categories and Subject Descriptors

H.4.m [**Information Systems Applications**]: Miscellaneous

Keywords

Analytics Flows; Translation; Language; SQL; Code Generation

1. INTRODUCTION

Enterprises today employ a variety of data repositories and processing engines to meet their needs for analytics. In addition to an SQL engine, a business might use an event processing engine, an information-extraction engine, a map-reduce engine, an ETL engine, and so on. This diversity of systems enables rapid development of new analytics, but also increases the management complexity. Even a simple analytics environment comprising a data warehouse and an ETL system might include dozens of analytic reports and ETL scripts. This is a management challenge and, as more repositories and processing engines are added to the mix, the complexity increases substantially.

In such a heterogeneous environment, analytics programs and computations span multiple execution engines and storage repositories and they typically form data flows, which we call hybrid flows. As we work with hybrid flows, we deal with issues related to maintenance, migration, development, and performance. Maintenance refers to program changes required when the underlying software is upgraded. Program migration is required when the underlying software is itself changed, e.g., from SQL database vendor X to database Y. Development of programs spanning multiple engines comes with several challenges. For example, to develop a hybrid flow, a typical practice is to create a workflow of individual tasks, each task directed at one engine; but then these tasks should be composed together. Or for debugging, it may be useful to decompose a large program as smaller, manageable tasks that are easier to investigate. Or we may have a program for an engine that is not available; e.g., a data cleansing script written for vendor X or for a system that is down for maintenance. Finally, once the analytic program is developed, it must be tuned for performance objectives. This may involve choosing the best engine to implement some operation, merging scripts, and so on.

To address these challenges, we created *BabbleFlow*. At a high-level, it inputs a set of analytic programs in one or more programming languages, optionally performs some transformation on the programs, and then outputs a set of analytic programs in one or more target languages. *BabbleFlow* represents computations at a logical, engine agnostic level. It inputs physical flows and converts them into a logical representation. With this approach, we can translate analytic programs from one language to another (or one software version to another), we can compose and decompose analytic programs to facilitate development and debugging, and we can optimize end-to-end programs for performance. Currently, we support a representative set of languages used for analytics, specifically, a declarative language, SQL, a procedural language, Apache PigLatin, and a flow, metadata language used in Pentaho PDI. But *BabbleFlow* has a general architecture and we can support new languages as needed. Note, to keep the problem tractable, we do not support general-purpose programming languages, just those with a fixed set of functionality.

BabbleFlow is a part of a larger system that manages hybrid analytic data flows [5]. But it can also stand as an independent tool for flow translation. In the past, we have described other system components, like a hybrid flow optimizer [3, 4], a collection statistics module (e.g., [3]), a flow execution scheduler (e.g., [5]), and so on. In this demonstration, the focus is on the internals of our flow translators and the flow composition and decomposition mechanisms, which enable engine independence. A detailed description of the technology demonstrated here and a discussion on the related work can be found elsewhere [1].

*Work done while with HP Labs, Palo Alto.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2594534.

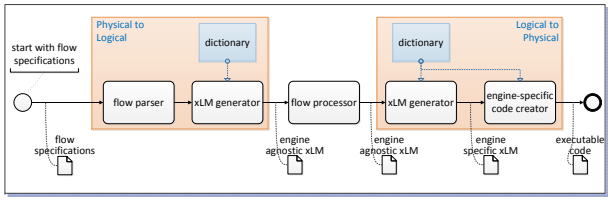

Figure 1: System architecture

2. DEMONSTRABLE FEATURES

2.1 System Overview

A system overview is illustrated in Figure 1. We first discuss two components of *BabbleFlow*, namely the xLM language and the dictionary, and then, we give a high level description of our system.

BabbleFlow models flows as directed, acyclic graphs (DAGs) and encodes them in our *lingua franca*, a data flow metadata language called *xLM* [2]. xLM describes flows, their operators (e.g., schemata, semantics) and data stores, and the interconnection among them. It also captures operational properties at the flow level and operator level, like resources used and physical characteristics, and can represent various levels of abstraction. In its current implementation, xLM is expressed in XML. Note that xLM is not a user language, but is intended for flow processing.

To enable translation from one language to another and from physical to logical flows and vice versa, we use an extensible dictionary of mappings between logical constructs and their valid physical constructs in the supported languages. Example logical constructs are operators, functions, expressions, and data types. In general, we may have 1-1, 1-n, and m-n mappings that map a logical construct to different physical constructs and vice versa. Figure 2 shows example, simplified entries for an operator (Project), a function (getDate), and a comparison operator (Not Like).

Figure 2: Example dictionary entries

The input to our system is a physical flow, which may be a hybrid flow spanning multiple engines. A 'physical to logical' module converts a physical, engine specific flow to a logical, engine agnostic flow. We parse the specifications of the physical flow (e.g., a script in a programming language or metadata that encodes the flow) and produce an engine agnostic xLM (a-xLM) representation of the flow constructs. To do this, we use a *dictionary of mappings* between logical constructs and their physical counterparts in the supported languages. During parsing, we also collect statistics and cost estimates for the flow and its operators. If the original flow comprises subflows (e.g., scripts) in more than one language, we translate and compose them all into a single logical flow.

Next, a 'flow processor' may transform the logical flow in some way; e.g., optimization, flow decomposition. Flow transformation can also be done manually by a flow designer; e.g., a user may determine the engine assignment, the code complexity (nested vs. decomposed flows), or the target programming language(s).

The 'logical to physical' module converts an engine agnostic flow (in a-xLM) into an engine specific flow (represented in engine specific xLM, s-xLM) according to the engine selections made either by a flow processor or a flow designer and using the xLM mappings stored in the dictionary. In some cases, a flow in s-xLM 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 s-xLM to executable code that can be dispatched for execution.

2.2 Flow Import

We created parsers to translate flows in PigLatin and/or SQL to a logical flow in xLM. First, we describe import of a single flow running on a single engine and next, we discuss the more general case of hybrid flow import and flow composition.

The import phase starts with a code script. If the flow runs on an engine that produces execution plans, we parse the *execution plan* of a flow instead of the actual program. We follow this method for SQL and PigLatin flows, as in both cases an execution plan is available. This abstracts us away from the intrinsic characteristics of each programming language and provides us directly with the flow structure and also, with extra useful information like cost estimates for flow operators (e.g., row cardinality, memory).

Figure 3 shows an example import for a PigLatin script. We probe the engine to get an explain plan (not shown in the figure) and parse it for identifying the engine specific, flow operators. We then use the dictionary to map each engine specific operator (e.g., LOFilter) to an engine agnostic operator (e.g., Filter) and express it in xLM (right part of Figure 3). In addition, we identify the semantics of each operator and convert it to a logical form. For example, for the filter, we analyze its filter condition, express it as an abstract expression tree (middle part of Figure 3), convert physical constructs like functions (e.g., ToDate) to logical, and create the appropriate xLM (a-xLM) representation for it (e.g., getDate).

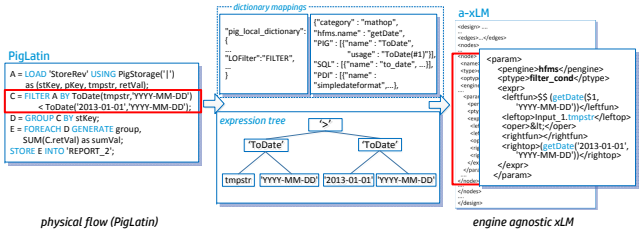

Figure 3: Example script import

An alternative way to parse the input flow is to modify the codebase of the engine. We did this for an open source ETL tool (Pentaho PDI), where we extended the code generation methods to generate xLM directly from the tool. Figure 4(left) shows an example import from PDI code: starting from a flow designed in PDI (top left corner), we produce engine agnostic xLM (bottom left corner).

BabbleFlow also handles n-m mappings where an operator in one language may be mapped to multiple operators in another; e.g., SQL Group By maps to two PigLatin operators, Group By and Foreach Generate. On the other hand, if an operator is not supported in an engine, then we do not consider shipping it to that engine.

Composition. For a hybrid flow having subflows running on multiple engines (e.g., Hadoop and SQL engine), we complement the above functionality with a composition mechanism. After we create engine agnostic xLM for each subflow running on a different engine, we identify *connect points* in the individual subflows; i.e., points where these subflows connect to each other.

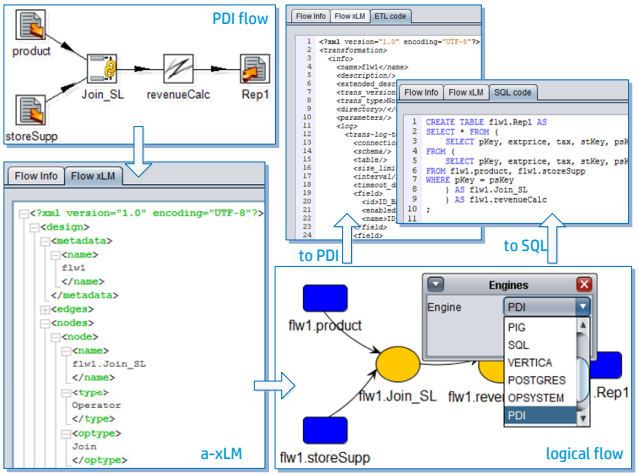

Figure 4: Code conversion from PDI to xLM and then, back to PDI again or to SQL

Connect point identification can be either a guided process, where a flow designer specifies the connections in the form of metadata, or a semi-automated process. In the latter case, the connect points are inferred based on an analysis of the input flows. For example, if two flows utilize an engine specific connector (e.g., an HDFS-to-database connector) connecting them to the same data storage, this storage 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 nodes between two flow graphs, based on the compatibility of the output schemata of the nodes of the first flow and input schemata of the nodes of the second flow. In this case, the flow designer can optionally validate the system choices. Advanced schema matching and mapping techniques are relevant to this task and are considered as an interesting future direction.

We support several flow connection types, like pipeline, blocking, or connection through a store point or a control point. The connect points are considered as operators in a-xLM and so enable modeling the entire computation as a single flow. This is a powerful feature of our system.

Figure 5 shows a composition of the example PDI and PigLatin scripts of Figures 4 and 3, respectively. In this case, *BabbleFlow* determined that the target data store of the PDI flow and the input data store of the PigLatin script are compatible; i.e., they have identical schemata. So, these serve as a connect point, which in turn was transformed to a pipeline connector between the flw1.revenueCalc_1 and C_LOFilter_2 operators (a pipeline operator can be omitted from the drawing as a syntactic sugar).

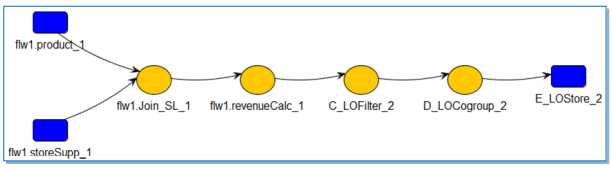

Figure 5: Example composition of the PDI and PigLatin scripts described in Figures 4 and 3, respectively

2.3 Flow Processors

A flow processor takes as input a logical flow in a-xLM, performs a transformation on that flow, and produces as output a second logical flow, which is functionally equivalent to the input, but perhaps with different properties. Note that the use of a flow processor is optional and is not needed for the flow language translation.

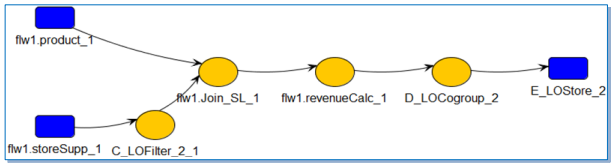

Figure 6: A flow transformation involving operator swapping

A flow optimizer is one example of a flow processor. Figure 6 shows a possible optimization for the flow of Figure 5. Since the two flows, PDI and PigLatin, have been composed to a single flow, the optimizer has an end-to-end view of the hybrid flow and can optimize it as a whole. In this example, the C_LOFilter_2 operator of Figure 5 has been pushed earlier in the flow.

Another flow processor might be used to decompose a long flow into subflows or to compose disconnected flows into a single flow. In doing so, the flow processor needs to alter connectors connecting the subflows running on different engines. For example, flow composition would remove connectors and flow decomposition would add connectors. Flow optimization may include function shipping, i.e., moving an operator from one engine to another, which 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 and possibly data shipping from the old engine to the new one. 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.

Besides flow processors, the engine assignment or flow transformation can be defined manually by a flow designer. The bottom, right corner of Figure 4 shows an example of how a user can choose an engine for a flow.

2.4 Flow Export

The export phase starts with a logical flow, for which engine assignment has been decided either by a flow processor or a designer. Then, *BabbleFlow* converts engine agnostic xLM (a-xLM) to engine specific xLM (s-xLM) using the dictionary of mappings and following a process similar to the physical to logical conversion.

Figure 7 shows a possible engine assignment for the flow in Figure 6. Different node colors denote different engine assignments. In this example, we maintained the original engine assignment, and thus, the flow comprises two subflows, a producer subflow shown in orange, assigned to PDI, and a consumer subflow, shown in yellow, assigned to PigLatin. These are linked by a single connect point. Note, that this example shows a case of function shipping as the filter operator in the input flow is moved from PigLatin to PDI.

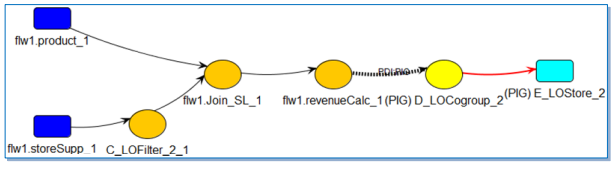

Figure 7: A multi-engine assignment for the flow of Figure 6

Next, we proceed with code generation. If the flow is hybrid, then we first *decompose* it into subflows each designed to run on a single engine, and then, we generate separate execution code (i.e., code script) for each subflow. We work similarly for same-engine subflows; e.g., subflows produced with flow decomposition. In both cases, *BabbleFlow* also generates the code needed for orchestrating the execution of the scripts (e.g., which flow starts first, when a subsequent flow should start, and so on).

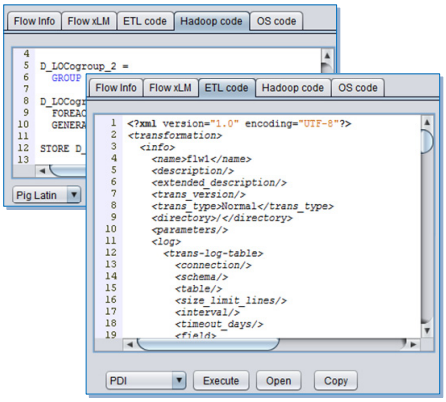

Figure 8: Code generation for the hybrid flow of Figure 7

The top, right corner of Figure 4 shows two possible single-engine translations for the PDI flow (shown in the top, left corner), one to SQL and one back to PDI. The code can either be sent for immediate execution or be scheduled for execution at a later time.

Figure 8 shows an example of hybrid code creation for the example of Figure 7. In this case, *BabbleFlow* creates two scripts, one for PDI and one for PigLatin, along with code for orchestrating their execution (under the OS code tab). The first tab, Flow Info, contains statistics and options for tuning the flow. Flow xLM contains the xLM code (here, s-xLM) for the flow. In the PigLatin snippet (Figure 8(back), lines 5 and 8) one may see two aggregation-related statements, whereas Figure 7 shows one, D_LOCogroup_2; this is an example 1-n case, where one logical operator converts to two physical operators, as discussed also in 2.2. Figure 9 shows an alternative case of generating nested SQL for the entire flow.

BabbleFlow has been tested to ensure translation correctness. The generated code may not be syntactically identical to the original (e.g., due to optimization or other flow transformations), but semantics are preserved and the code functionality is unchanged. It is worth noting that currently our orchestration code does not resolve failures that may occur (e.g., a table no longer exists, connection or machine failures, runtime errors), but to the extent possible, *BabbleFlow* catches these errors, reports them back to the user, and halts the execution of pending subflows. On the other hand, errors such as incorrect dictionary mappings are harder to catch. As with proving a compiler is correct, it is hard to formally prove correctness properties. Toward this direction, we provide the user with a crude test-and-learn mechanism to validate a mapping through experimentation with example data. More details on correctness issues can be found in [1].

2.5 Implementation Details

BabbleFlow was developed in HP Labs. Its core engine is implemented in about 25K lines of Java 1.7 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, we get execution plans for SQL by probing the engine using JDBC and for PigLatin, by using an external library, PigServer, without connecting to the engine. We also modified the open source codebase of PDI to make it xLM aware. We provide a Java based GUI for importing, viewing, modifying, tuning, debugging, and exporting analytic flows.

BabbleFlow is a component of the Hybrid Flow Management System (HFMS) [5]. Other HFMS components include a hybrid flow optimizer (e.g., to choose engines and implementations for a flow), a flow workload manager, a flow executor, etc. Currently, we are extending *BabbleFlow* to support additional languages.

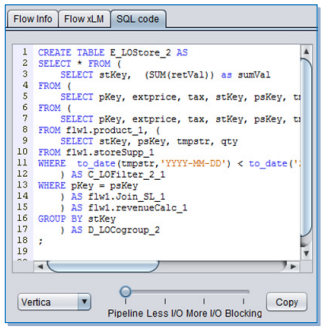

Figure 9: Single engine code for the hybrid flow of Figure 7

3. DEMO

We will demonstrate *BabbleFlow* features using TPC-DS and TPC-H queries, all implemented in SQL, PigLatin, and PDI. These are representative examples of analytical scenarios and have a wide range of execution times and complexity, and so they provide many options for illustrating features. We will also have custom flows combining ETL and analytical logic and having their subflows implemented either in one or in two languages. For the demonstration, we will use a SQL engine, Hadoop, and an ETL tool. We will use the following scenarios to demonstrate the usability of *BabbleFlow*.

Code migration. A new user of database engine db-A may have legacy SQL scripts written for a different database version or release, db-B. Or a new db-A release might include data cleansing functionality that a user used to perform within an ETL engine. *BabbleFlow* can migrate the user's legacy scripts to db-A.

Function shipping. Existing technology allows connecting from one engine to another (e.g., through connectors), but this only serves as a basic interconnection infrastructure. *BabbleFlow* offers an extra tool: it enables *function shipping* too and so, operations can be moved closer to the data, avoiding movement of large datasets. An example scenario involves a user who has PigLatin code for HDFS data and wishes to process this data within db-A instead of using an HDFS-to-db connector. *BabbleFlow* can translate these scripts.

Debugging. A db-A user may have a complicated, nested SQL query that is hard to read and comprehend. Or, conversely, the same user may have a single computation written as multiple SQL or PigLatin scripts linked by temporary tables or files, which, to improve performance, should be rewritten as a single query. *BabbleFlow* can compose and decompose these scripts.

Hybrid flow management. Having the ability to manipulate hybrid flows at a higher level, as a single entity, can be a key differentiator in modern, multi-engine environments. Being able to handle the different pieces of a hybrid flow as one, assists the design, optimization, debugging, and execution of a flow. For example, we can decompose at runtime long running queries (which may monopolize computing resources like CPU, memory) to smaller queries interconnected through temporary storage; and this can often result in more robust resource sharing across a workload.

4. REFERENCES

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

[2] A. Simitsis and K. Wilkinson. The specification for xLM: an encoding for analytic flows. Technical report, HP Labs, 2014.

[3] A. Simitsis, K. Wilkinson, and U. Dayal. Hybrid analytic flows - the case for optimization Fundamenta Informaticae, 128(3):303-335,2013.

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

[5] A. Simitsis et al. HFMS: Managing the lifecycle and complexity of hybrid analytic data flows. In ICDE, pages 1174–1185, 2013.