HFMS: Managing the Lifecycle and Complexity of Hybrid Analytic Data Flows

Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, Meichun Hsu

HP Labs
Palo Alto, USA
{firstname.lastname}@hp.com

***Abstract*— To remain competitive, enterprises are evolving their business intelligence systems to provide dynamic, near real-time views of business activities. To enable this, they deploy complex workflows of analytic data flows that access multiple storage repositories and execution engines and that span the enterprise and even outside the enterprise. We call these multi-engine flows hybrid flows. Designing and optimizing hybrid flows is a challenging task. Managing a workload of hybrid flows is even more challenging since their execution engines are likely under different administrative domains and there is no single point of control. To address these needs, we present a Hybrid Flow Management System (HFMS). It is an independent software layer over a number of independent execution engines and storage repositories. It simplifies the design of analytic data flows and includes optimization and executor modules to produce optimized executable flows that can run across multiple execution engines. HFMS dispatches flows for execution and monitors their progress. To meet service level objectives for a workload, it may dynamically change a flow's execution plan to avoid processing bottlenecks in the computing infrastructure. We present the architecture of HFMS and describe its components. To demonstrate its potential benefit, we describe performance results for running sample batch workloads with and without HFMS. The ability to monitor multiple execution engines and to dynamically adjust plans enables HFMS to provide better service guarantees and better system utilization.**

I. The Emergence of Hybrid Flows

In the beginning, before the star schema, before the data warehouse, before ETL engines, business analysts struggled to gather up-to-date information on the current state of an enterprise. Hand-coded reports were easily broken when the underlying production systems changed; creating new reports was time-consuming and error-prone. The introduction of back-end data warehouses vastly improved the situation by providing a common schema and framework for moving data across the enterprise. This insulated business users from changes to IT infrastructure. Although costly to design and build, this infrastructure soon paid for itself through higher productivity and more accurate business views. But we are entering a new era of instability. Enterprises must respond more rapidly to customers. They must reduce the latency between business event and business action and must extract data from new information channels, e.g., social media. A back-end data warehouse is centrally-controlled and, thus, slowly evolving (at best, monthly changes). It simply cannot keep pace. So, individual businesses within an enterprise are forced to build "shadow-IT" to utilize new technologies and remain responsive.

For example, a business unit might leverage a public cloud to deal with infrequent, large batch jobs; e.g., content analysis of scanned documents. A product division might use a map-reduce engine for log file analysis. Such computation is rarely done in isolation. Typically, the results are merged with analysis from other data sources forming a graph of operators and a data flow that spans multiple storage repositories and execution engines. We refer to these multi-engine flows as *hybrid* flows. For example, a consumer product company might use a hybrid flow for real-time feedback on new products. It might use a map-reduce engine to compute customer sentiments about products from a Twitter stream. Separately, a query over an operational data store would aggregate recent sales data. The results of these flows are then joined to correlate sales to product sentiment, and thus, evaluate product launches.

In current practice, hybrid flows are created using a custom workflow of scripts and programs that perform step-wise analysis on each system and transfer result data to the next step. Although there are sometimes direct connectors between engines (e.g., ODBC connectors between ETL and database engines), in general, there is no unifying view or management of the entire hybrid flow. The individual systems may perform local optimizations for their step of the flow, but local optimizations at one engine may conflict with that of another. Any end-to-end optimization of the complete hybrid flow must be done manually by the flow designer. Hence, designing and implementing a correct, efficient hybrid flow is difficult because such flows involve many independent computing systems, interfaces, and data sources. Also, the underlying computing infrastructure undergoes periodic change that may require corresponding updates to existing flows. And we observe that flows are often batched into workloads or classes such that the batch or class has specified service level objectives. To the best of our knowledge, existing workload management tools are engine-specific; there are no tools to manage workloads of hybrid flows. Thus, as hybrid flows become more common, the current, custom, manual approach cannot scale.

We are dealing with the consequence of one-size-does-not-fit-all; i.e., use the most appropriate engine for a task. For large problems, this means multiple engines. To deal with this complexity, one approach is a programming/scripting language with connectors to import data from various engines; e.g., multi-database systems, parallel execution frameworks like DryadLINQ, HadoopDB, Stratosphere, etc. (see Section VIII).

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in IEEE ICDE 2013.

We introduce a different approach, a Hybrid Flow Management System (HFMS). Like other multi-engine systems, HFMS provides a flow language. But rather than unidirectional connectors, HFMS treats multiple engines as peers. Data shipping and function shipping occur in either direction. Additionally, HFMS optimizes for multiple objectives across engines; e.g., where fault-tolerance is important for part of a flow, a map-reduce engine could be used. Also, HFMS includes a workload management component to deploy, monitor, and modify flow execution. We know of no other system that treats hybrid flows as first-class objects.

In this paper, we first present our architecture for HFMS and describe its components. Then, we motivate the need for this technology by showing how the ability to optimize *and* manage flows across multiple engines enables administrators to meet workload and corporate objectives.

No commercial RDBMS vendor would consider selling a relational database engine without an optimizer and a workload management tool. Customers demand it. We see the same need emerging for hybrid flows as they become more common and more critical to daily operations. An HFMS gives enterprises a central point of control to create, deploy, and manage hybrid flows.

II. Engines, Operators, Flows, and Workloads

First, we define concepts central to the design of HFMS.

Execution engines. At a high level, the computing infrastructure of an enterprise may be viewed as a collection of execution engines. Each engine supports a number of object types (e.g., files, tables) and a set of operators over those types. Most engines support composition of their operators where the output of one operator (the producer) becomes the input to another (the consumer). We refer to this as a *single-engine* flow. Some engines provide persistent storage for objects and we refer to these as storage engines or repositories. Additionally, engines require methods to import and export objects. These methods may provide a direct connection to some other engine (as an ETL engine reads data from a database engine) or they may simply access files or data streams.

Some engines include tools for defining flows, optimizing flows, and managing their execution. HFMS is intended to complement such tools, not to replace them. For example, compared to HFMS, the internal optimizer of a database engine has more knowledge of system internals and so, it would produce a better execution plan (e.g., join order). However, the database engine cannot know that a flow it is executing is part of a larger flow that spans other engines. This is where HFMS can help. HFMS is itself an engine that operates on flows. As described later, it determines how best to partition a flow into single-engine sub-flows, dispatches the sub-flows for execution, and orchestrates their execution to satisfy dependencies.

Operators. Within a single engine, an operator may have multiple implementations; e.g., different join implementations in a DBMS. Additionally, an operator may be supported on multiple engines; e.g., most engines support a filter operator

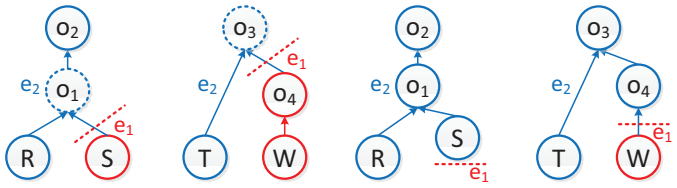


Fig. 1. Hybrid operators

to select data objects based on some condition. Operator polymorphism complicates the flow optimization. For example, if an operator o can run on engines e_1 and e_2, then, we need to consider the cases of executing it on e_1, e_2 or both (i.e., we do a parallel partial or full computation and then merge the results). In addition, a composite operator may be decomposable into smaller units where some computation runs on one engine and some on another. For example, a simple filter operation with a condition A *and* B could be split into two consecutive filter operations, one for each conjunct. We describe such a decomposition in the context of hybrid flows elsewhere [1].

For hybrid flows, a challenging case is hybrid operators. A *hybrid operator* is any operator that runs on one execution engine but consumes input produced by an operator on a second engine. The case of a unary, hybrid operator is not interesting; this can be seen as a simple data shipping operation. The case of n-ary, hybrid operators is more challenging. In the rest, we consider hybrid operators with more than one input and where at least two inputs are produced by different engines. These inputs can be either data stores or other operators.

Figure 1(a) shows a hybrid operator o_1 (shown with a dashed line) that runs on engine e_2 with inputs from data stores R and S placed on engines e_2 and e_1, respectively. Figure 1(b) shows a hybrid operator o_3 with inputs from a local data store T and from another operator o_4 running on a different engine. There are choices for configuring the execution of hybrid operators. We can engage the engines involved and use either data or function shipping. Figure 1(c) shows a possible solution when the remote input of o_1 is a data store: we ship the data from S into engine e_2 (data shipping). Similarly, when the input of o_3 is a remote operator, say o_4, we consider shipping o_4 to e_2, as shown in Figure 1(d) (function shipping). Observe that after these operations, o_1 and o_3 are no longer hybrid operators (thus, shown with regular lines).

We discuss shortly that HFMS supports and can execute a number of operators as well as control constructs for flows (e.g., sequence, conditional, iteration, etc.). For example, HFMS may take on the execution of o_1 of the previous example, by orchestrating the data pipeline from engines e_1 and e_2.

Flows. A flow describes the movement of data sets from a set of source engines to a set of target engines through a series of data transformation operators. HFMS provides a high-level view of flows. Data can be of any type (e.g., text, tuples, documents, images) supported by the underlying engines and data stores represent any static or dynamic source of data (e.g., tables, files, sensors, web portals). The set of HFMS operators

goes beyond those typically considered in databases and ETL engines, including, for example, text and data analytics, machine learning, image processing operations. Extensibility is provided through user-defined functions.

The design of flows in HFMS is physio-logical, meaning that the initial flow design may be agnostic to a physical implementation. Data sets need not be bound to a data store and operators need not be bound to an execution engine. Therefore, they can be optimized at a logical level. To optimize a flow, HFMS handles flows as graphs, namely *flow graphs*, and considers the choice of execution engines, the cost of operators, the sizes of data sets, and the optimization objectives. HFMS uses many optimization strategies, such as operator swap, parallelization, recovery points, function shipping, data shipping, decomposition, etc. HFMS internally represents a flow as a directed graph to be executed on one or more execution engines. The details of the internal representation are not relevant here. But, note that we require a flow graph to be acyclic with at least one start node and one or more terminal nodes. Prohibiting cycles is a somewhat arbitrary but practical requirement. Most execution engines lack support for cycles within a flow. Instead, iterations are performed by repeatedly running the flow and checking termination conditions.

The result of optimization is an execution plan which we also refer to as a physical flow. In an execution plan, operators and data sets are bound to specific implementations and specific engines. The execution plan may take several forms. It could be a single flow over a single engine. Or, it may be useful to partition a flow into sub-flows, e.g., to reduce complexity of a large flow or to better control of resource usage. We use *multi-flow* to refer to a set of flows used to implement a larger flow. For example, consider an SQL query that uses a common view in several places. It may be more efficient to materialize that view in a temporary table (first sub-flow) that is then referenced in the remainder of the query (second sub-flow).

When a multi-flow uses multiple engines, we refer to it as a *hybrid flow*. For example, if a hybrid operator has no direct connection to the engine that produces one of its inputs, the flow must be broken at that point into separate sub-flows, one for each engine. Hence, for multi-flows and hybrid flows, HFMS must track dependencies or connections among flows.

The distinction between single flows and multi-flows is common. ETL engines separate flow definition from flow scheduling. In map-reduce engines, a job comprises a map step and a reduce step, each, in effect, separate sub-flows. More interestingly, in HFMS, a PigLatin or Hive script for example is considered a single job (flow) comprising a number of individual map-reduce steps (sub-flows). Other systems would treat the individual steps independently and not recognize them as part of the same job.

Workloads. Given a flow execution plan, HFMS may deploy the plan on one or more execution engines and orchestrate its execution. However, there are two issues with simply running the plan from the optimizer. First, the state of the computing infrastructure may have changed between the time the plan was produced and the time it is executed. For example, an execution engine may have failed or there may be contention for resources. Second, it is common for a flow to be a member of a set of flows with some common characteristics. The set is treated as a workload for administration and management. Example workloads may include a set of weekly reports or a set of nightly ETL flows to refresh a warehouse. Or a workload might comprise all flows submitted by a class of users, e.g., flows submitted by analysts might be one workload, flows from administrators another workload, flows from customers (e.g., purchase orders) might be a different workload, etc.

Workloads, like individual flows, have objectives which are typically specified as service level objectives (SLO). For a batch workload, an objective might be a completion deadline. For an interactive workload, an objective might be that 80 percent of the flows must finish in less than 10 seconds. The SLO may specify penalties for not meeting the objectives.

To support workloads, HFMS needs additional capabilities. It must associate flows with workloads. It must enable the definition of policies to achieve an SLO. A policy specifies actions to take when some condition occurs. HFMS must monitor the current system load and the service level for a workload and, if necessary, apply policies to improve service for a workload. We discuss all these shortly.

III. HFMS Architecture

The HFMS architecture has three layers: Interface, Optimizer, and Executor (Figure 2). These provide for flow design, optimization, and execution dispatch and monitoring. HFMS flows execute primarily on independent execution engines and storage repositories. By independent, we mean an engine may be accessed through other paths beside HFMS. Within the executor, HFMS has its own intrinsic execution engine but it is mainly intended for flow orchestration. The optimizer will push data-intensive operations to the independent engines.

Interface. The interface layer accepts a flow definition in a variety of representations. The HFMS designer GUI allows users to interactively create or modify a flow. Alternatively, flows may be written in a declarative language (e.g., SQL, Pig, Hive) and imported through an API. And, HFMS provides connectors to import flows from external design editors (e.g., an ETL or a workflow tools). The HFMS plan writer takes a flow from one of these interfaces, checks its validity, and translates it into an internal, machine-interpretable form and passes it to planner module in the optimizer.

Optimizer. The optimizer generates a functionally equivalent flow graph optimized for user-specified objectives. This includes partitioning the flow into subgraphs for different execution engines. The planner module considers alternative execution plans and estimates their cost. For this, each flow operator has associated cost models, a different model per implementation and engine, derived using various techniques. We discuss optimization issues in Section V-B. The optimal plan (i.e., the least-cost relative to the objectives) is sent to the executor.

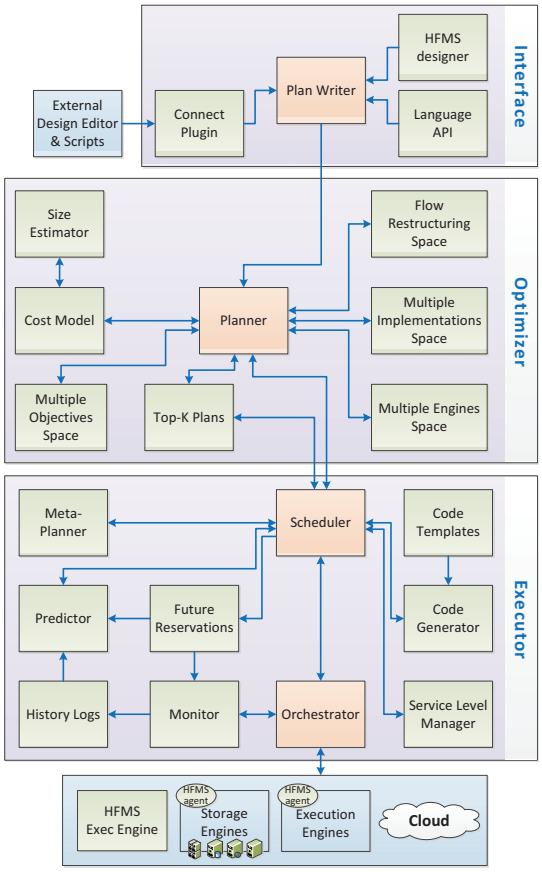


Fig. 2. HFMS architecture

Executor. The executor performs several tasks. It schedules flows for execution, generates executable code, dispatches flow fragments to execution engines, and monitors flow execution. The scheduler decides when to run a flow based on its requested start time, its required resources compared to those available and the flow's workload or service level class. The scheduler handles immediate flow execution and also periodic (hourly, daily, weekly, monthly) flows and can schedule flows far into the future. When a flow is ready to run, HFMS checks that the optimizer's plan is still valid. If needed, it finds an alternative plan based on the current system state. In general, a flow comprises a number of flow fragments each assigned to an execution engine. For each fragment, the executable code (e.g., SQL, PigLatin) is generated along with the orchestration code; i.e., control flow code, needed to order and link flow fragments. The flow is then dispatched to the various engines for execution. A monitor observes the system states and tracks the progress of executing flows.

IV. Interface

We provide an API to enable users and other applications to communicate with HFMS. Internally, a flow is encoded in terms of xLM, our proprietary language for expressing data flows. A flow can be either created in or imported to HFMS.

A GUI tool can be used to create a new flow from scratch. The GUI is an extension of Pentaho PDI (a.k.a. Kettle) [2], which we modified to encode flows as xLM to connect with our platform. Alternatively, an existing flow may be imported from some other design tool (e.g., an ETL or workflow engine) or scripts (e.g., SQL, Apache Pig). All these forms are translated into xLM. And as we discuss shortly, they can be translated back to one of the supported execution forms, when flow optimization and processing has terminated.

xLM captures structural information, design metadata (e.g., functional and non-functional requirements, physical characteristics like resource allocation), operator properties (e.g., type, schemata, statistics, engine and implementation details, physical characteristics like memory budget), and so on.

For translating a flow to xLM, we parse the input flow, identify operations and data stores and map them to a library of operators supported in HFMS. We support a fairly large number of operators (more than a hundred) and it is relatively easy to register a new one using our wizard for monitoring, modifying, and registering operators.

V. Optimizer

A. Cost model

A fundamental aspect in flow optimization in HFMS is the accurate and effective computation of flow cost. Inaccurate costs lead to suboptimal plans. Ineffective cost computation slows down the entire system. The core elements for the cost model are: (a) operator cost, and (b) statistical summary of operator's output. Both depend on statistical summaries of the data. A detailed discussion of these summaries is omitted for space considerations.

The operator cost involves measures like CPU, memory (e.g., buffer sizes), I/O, and communication costs. Each implementation comes with a different cost and the same holds for operator incarnations on different engines. In most cases, the operator developer or provider does not disclose a cost formula for the operator (and the source code is not available either). Then, we need to run a series of micro-benchmarks to study the operator's behavior under different configurations [3]. As an alternative method, we build prediction models using machine learning techniques over execution logs, in the spirit of [4]. Based on such methods, we can build cost models for operators in different environments. As an extra but optional step, we allow users to run their flows with a sample of their data and we use the statistics obtained to fine-tune our cost models before flow optimization. The sampling method is a parameter to the system, which intrinsically supports a few basic sampling methods like Reservoir sampling and it can be easily extended to support others too.

B. Flow optimization

The hybrid flow optimization problem can be viewed as a difficult search problem. As with traditional query optimizers in relational systems [5], HFMS needs: (a) a search space, (b) a cost estimation technique for each plan in the space, and (c) an enumeration algorithm that searches through the space.

Search space. In HFMS, the search space of possible execution plans for a flow is multi-dimensional. At minimum, it

involves a flow restructuring space, a multiple implementation space, a multiple engine space, and a multiple objective space.

Starting with a flow, we apply several transitions to produce equivalent, but restructured flows. Example transitions include operator (or group of operators) reordering based on cost based heuristics or a generalization of interesting orders [6], partitioning parallelism, replication, and so on. An operator may be implemented by more than one algorithm. Thus, another dimension of the search space involves different implementations of the same flow execution plan. In addition, a complex or composite operator may be decomposed into a set of simpler operators and vice versa. For example, a sentiment analysis operator may be executed either as a single operator or as a collection of operators such as tokenization, part-of-speech tagging, attribute extraction, and so on [1]. A third dimension relates to the multiplicity of engines. An operator may have implementations on more than one engine. Thus, there are several ways to split a hybrid flow such that each fragment runs on a different engine. Possible transitions include data and function shipping. A flow may be optimized for performance or for other objectives like fault-tolerance, freshness, accuracy, and so on. Thus, multiple objective optimization adds a fourth dimension in the search space.

Search techniques. We use various techniques for exploring a set of alternative execution plans as specified by the multidimensional search space and finding the cheapest one as determined by the cost model and an objective function. An example, but not always practical, technique is a dynamically pruning, exhaustive search algorithm. Starting from an initial flow, this method constructs all alternative plans by iterating on the number of possible transitions (e.g., restructuring transitions, data or function shipping) and pruning plans known to be suboptimal. More elaborate algorithms are used, too. Although the optimization problem at hand is more complex (e.g., more complex operators, multiple engines and objectives) than traditional query optimization, we have also built upon some of the well-know query optimization techniques like rule based strategies [7] and heuristic techniques used for ETL flows [8].

C. Multiple-objectives and top-k plans

One of the core challenges in HFMS involves the multiplicity of objectives. Frequently, the optimization objectives are conflicting, as for example, a fault tolerant flow does not typically have the optimal performance. Thus, although the other spaces can be explored as described before, navigating through the multiple objectives space trying to resolve trade-offs and satisfy conflicting objectives requires a different approach. This search problem is formulated as a maximum vector problem (a.k.a. skyline computation) [9], where the solution should maximize a monotone scoring function representing an optimization (functional or non-functional) objective.

As we work toward satisfying multiple objectives, we identify a number of *interesting* states, each of which optimizes an objective or a specific configuration and we store them in a top-k list of plans (as shown in Figure 2). As an example, assume that we have two possible execution engines, a parallel database engine and a map-reduce engine. Based on the flow characteristics, the cost model, and the objectives, the optimizer may determine that the parallel database engine is the best choice for executing a flow. That means, that the least-cost state, along with the next-best states, would suggest using the database engine. However, in a situation where the database engine is otherwise overloaded, instead of waiting for executing the flow, HFMS will try to use the second-best engine if it predicts that this might be a faster solution. In this case, we would need to use the least-cost state involving the map-reduce engine, although this state may be "far" from the best one in the state space. Hence, the top-k plans do not necessarily correspond to the k cheapest plans, rather the top-k list contains a *sample* of the multidimensional search space that covers the given computing environments in the best possible way.

ID	cost	size	RPs	PAR	score	best-for
441	2295660	29	0	4	577	performance (pdb-n)
512	2295935	32	2	2	592	performance (pdb-d)
624	2303204	38	0	4	603	performance (mr)
13	2350110	21	0	0	661	maintainability
1779	3743899	52	15	2	1332	recoverability (hard)
951	2295890	32	3	1	484	recoverability (soft)
348	2350669	27	6	0	433	globally optimal
1233	2296219	35	6	1	439	optimal in all but maintainability

TABLE I
SOLUTIONS FOR VARIOUS OBJECTIVES

An example set of possible solutions for a given flow is shown in Table I. This table illustrates the best solutions found for a given flow. For example, solution ♯441 is best for performance (achieves better partitioning, PAR, too). However, solution ♯951 has only a slightly higher cost yet offers better fault tolerance (note three recovery points, RP). Also, regarding performance, the top-k list stores the least-cost solution per different combination of engines. For example, ♯441 and ♯512 are the best solutions for running in a parallel database engine (pdb) as a nested (single flow) and decomposed (multi-flow) forms. ♯624 is the suggested flow for executing in a map-reduce environment. A global score is computed as a weighted aggregate of all objectives.

VI. EXECUTOR

The executor has two primary tasks. First, it takes flows from the optimizer and schedules them for execution. Second, when a flow is ready for execution, it dispatches the flow to the engines and monitors its execution. These tasks are performed by the two principal executor modules: scheduler and orchestrator.

A. Scheduling and alternative plans

A flow arriving at the scheduler optionally includes a desired start time window and a class identifier. The start time is used to specify execution at some time in the future or for periodic execution (nightly, weekly). Otherwise, a flow is queued for immediate execution. The class identifier associates the flow

with a workload either as part of a batch or through a service class. Service classes and workload definitions are maintained by the service level manager. It allows administrators to define policies to support service level objectives and tracks the quality of service for each objective.

The future reservations module queues flows for future execution. It maintains a calendar that tracks the scheduled work for engines at each point in the future. A user may also directly request flow execution through our reservation assistant, CloudAlloc [10]. The reservations module assigns time slots in a way that optimizes for objectives set by the administrator (e.g., response time, power consumption, machine utilization, provider profit, consumer cost). In addition, a predictor module continuously correlates past runs with future reservations and reevaluates the accuracy of the expected flow resource requirements (e.g., searches for cases of possible over-provisioning, patterns leading to congestion, and so on).

The scheduler moves a flow to its ready queue based on its required resources, the known execution environment availability, and its execution time window. Example scheduling policies are described in more detail later. However, a ready flow is not immediately prepared for execution. The plan must first be validated. The optimizer has previously determined an optimal plan for a flow, but there is no guarantee that plan remains optimal at execution time. For example, the optimizer may know that engine e_2 is faster than engine e_1 (it may have faster implementations or more resources) and so it favors plans that use e_2. After starting a number of plans, we may reach the maximum computing capability of e_2 causing ready plans to wait. In such a case, HFMS may consider an alternative plan.

Validation and replanning is done by the meta-planner. As discussed, during initial optimization, HFMS retains the top-k important plans. Thus, rather than dynamically re-optimizing a flow, HFMS can choose another plan from its top-k list based on the current state of the computing environment. If none is found, then we try with different awaiting flows and at the end, based on cost computations and comparisons, we may re-optimize a flow, e.g., to avoid a particular engine. The meta-planner also finds opportunities for multi-flow optimization. For example, if flows f_1 and f_2 are both ready for execution and they contain some identical flow fragments over the same data, then it is possible to execute f_1 along with f_2 even if they are not initially assigned to the same engine for execution.

Given a valid plan, the scheduler generates executable code for the execution using the code generator and the associated code templates. This results into a multi-flow for one or more engines, along with control structures to sequence its execution. This is then passed to the orchestrator for execution.

B. Orchestration and monitoring

The orchestrator module organizes flow execution. In general, a flow comprises a number of flow fragments each assigned to an execution engine. For each fragment, the code generator generates the executable form required by an engine

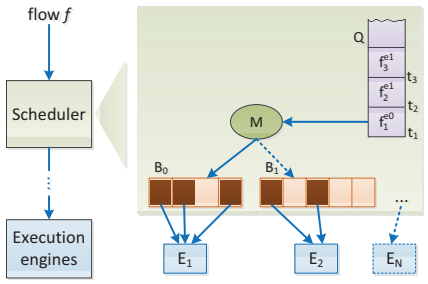


Fig. 3. Abstract functionality of scheduler

(e.g., SQL, PigLatin). The code generator connects to a template library that stores code templates for the supported execution engines. Flow fragments that run on different engines are linked with inter-engine connectors. The HFMS intrinsic execution engine may be used for orchestrating the flow fragments or simple data manipulation such as combining results but it is not intended for data-intensive processing.

At the end, the orchestrator dispatches code fragments to the execution engines, which actually execute the flow. HFMS installs a light-weight, low overhead agent on all compute and storage nodes to collect real-time statistics on resource usage. A monitor observes HFMS operation and keeps the actual usage in history logs. The monitor also follows flow execution, reports on its progress and intervenes, if necessary. In case of a failure, the monitor reports the error.

C. Scheduler

In this subsection, we present more details on scheduling. We first describe a basic scheduling algorithm and then present some actions that the scheduler may use to meet service level objectives. To simplify the discussion, we assume single engine flows and we use the term scheduler throughout this section rather than identify the specific module responsible for some functionality (e.g., meta-planner, predictor, etc.). Hybrid flows are considered at the end of the section. We also assume all flows are in a single service class.

As a starting point, we consider the simplest scheduling algorithm which is to limit MPL at each execution engine. We call this the baseline policy. For now, let us assume a fixed MPL. The scheduler uses two data structures: a ready queue and a set of run vectors, one for each engine. Figure 3 illustrates the scheduler's functionality. Each flow f that is ready to execute enters a queue Q that orders flows by their arrival time or requested execution time, t. The run vector for an engine, B_e, identities the flows currently executing on that engine. There is one slot per executing flow, so the MPL for an engine determines the size of its run vector. A discussion on MPL determination falls out of the scope of this paper; however, there are several past reports on this topic (e.g., for DBMS [11]).

When the run vector for an engine has a free slot, the scheduler searches in order through the ready queue for a flow that uses the engine. If one is found, the flow is dequeued and after plan validation and code generation, the flow is dispatched to the engine.

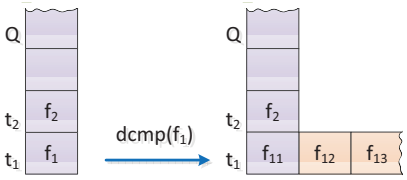


Fig. 4. Schematic representation of flow decomposition

Next, we describe some example scheduling actions. The set of possible actions and policies is large and depends on the objectives to be satisfied. Rather than be comprehensive our goal here is to show that even with simple policies, an external system like HFMS that sits on top of a plurality of execution environments may significantly improve the orchestration and execution of analytic flows by reaching opportunities not currently handled by existing systems and engine-specific optimizers. In Section VII, we demonstrate representative execution results for some policies.

1) Decomposition action: The decomposition action takes a single flow and produces an equivalent multi-flow in which the sub-flows are executed sequentially. For example, given a flow implemented as nested SQL, decomposition would create a sub-flow for each nested sub-query, where the result of the sub-query is saved in a table that the can be read by the parent sub-query. Each sub-flow is run independently, i.e., in a separate session. We acknowledge that, if a data store is modified between sub-flows, data inconsistencies may occur (e.g., non-repeatable reads). Therefore, for flows where this is an issue, decomposition would not be used.

The intuition behind decomposition is that executing multiple, smaller flows should cause less resource contention than a single, large flow that uses a large fraction of an engine. The multi-flows may perform more I/O than the single flow due to storing of intermediate result. But, overall an engine may have more consistent performance for other flows because spikes in resource usage are eliminated (see also Section VII).

A policy for using the decomposition action might be as follows. First, we evaluate the resource requirements of the flow. If the flow cost exceeds a budget θ, then, we decompose the flow. Decomposition is performed by the optimizer with a particular strategy.

Decomposition returns a set of sub-flows S_f. These sub-flows are put back into the Q as described below (see also Figure 4). Suppose Q initially contains two flows f_1 and f_2 to be executed on an engine e and a policy chooses to decompose f_1 into the sub-flows $S_{f_1} = \{f_{11}, f_{12}, f_{13}, \ldots\}$. Figure 4 illustrates this process. Observe that sub-flows S_{f_1} are not put immediately back to the list. We keep them in a buffer attached to Q. The first sub-flow, here f_{11} is executed immediately on e. If f_{11} terminates before the arrival time t_2 of f_2, the next sub-flow, f_{12}, is executed on e too. Otherwise, we execute f_2, and any other pending flow that has arrived before f_{11} finishes. We continue similarly until all flows have been processed.

2) Flow shipping action: The flow shipping action takes a flow to be run on one engine and reassigns it to run on a different engine. The intuition here is that overall service may be improved by off-loading work from a heavily used engine onto a lightly used engine. This action could be employed by various policies; e.g., only ship flows above a certain cost threshold, do not ship flows if the cost is higher on the new engine, etc. The flow shipping action first identifies all engines, say Φ, that are capable of executing the flow; i.e., all operators of the flow are supported in the engine. Changing the engine affects the flow cost, where now we should also account for the function shipping cost for operators. Cost is also affected by whether the involved data reside in the new engine too. Even if it does not, it is still possible to use the new engine, but we also need to consider the data shipping cost. (Note that in practice, this case is rare, since the data shipping cost typically dominates the total flow processing cost.) For more information about function and data shipping costs, we refer the interested reader in a previous report [1].

Identifying the next-best available engine is realized by a function $top(e_x), e_x \in \Phi$. This function may use the top-k plans for the flow produced by the optimizer (see Figure 2) to find the best plan that uses an available engine. Or it may ask the optimizer to reoptimize the plan for the new engine.

3) Scheduling hybrid flows: The previous discussion assumed single-engine flows. Hybrid flows can be handled with minor changes. Let $f_h^{e_x}$ denote a flow h to be run on engine e_x. A hybrid flow, say f_h: $f_{h1}^{e_1} \rightarrow f_{h2}^{e_2}$, can be considered as a combination of multiple sub-flows, $f_{h1}^{e_1}$ and $f_{h2}^{e_2}$, with dependencies. Similarly, a multi-flow, say f_m: $f_{m1}^{e} \rightarrow f_{m2}^{e}$, can be considered as a combination of multiple sub-flows, f_{m1}^{e} and f_{m2}^{e}, with dependencies. Thus, each sub-flow gets its own position in Q; but then, we look at the dependencies before the flow is executed.

Between two sub-flows there are two types of dependencies. A sequential dependency means the first sub-flow must terminate before the second sub-flow can begin. For example, sub-flows resulting from decomposition generally have a sequential dependency to ensure that the intermediate result tables are written before they are read. A parallel dependency means that the two sub-flows must start together. This dependency is used when a producer sub-flow pipes data to a consumer sub-flow. The scheduler must honor these dependencies when choosing to run a flow. For example, if two sub-flows have a parallel dependency, the scheduler cannot run them until their targeted engines have free slots in their run vectors.

VII. Use Case - Batch Workloads over Map-reduce and RDBMS Engines

In this section, we experiment with some sample policies that use the actions presented in Section VI-C to control the execution of a workload. We assume a set of flows that have been imported into HFMS as scripts and optimized by the optimizer in previous stages, as described before. Here, we focus on their execution. To demonstrate the potential of HFMS, we compare the performance of batch workloads running under the baseline scheduling policy with the same workloads running with policies for decomposition and flow

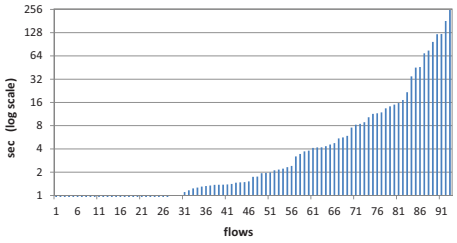


Fig. 5. TPC-DS flows execution times

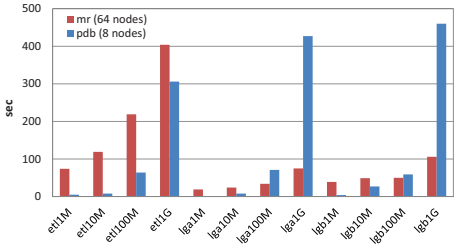


Fig. 6. ETL flows execution times

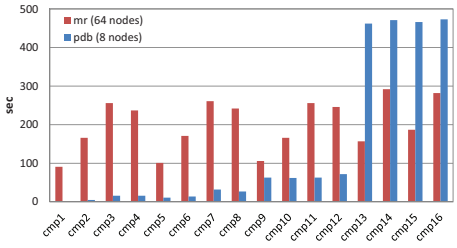


Fig. 7. Campaign flows execution times

shipping. In each experiment, HFMS is presented with a batch of flows and is responsible for dispatching the flows to the execution engines.

Each flow has been optimized to run on a single engine. We assume all flows in a batch arrive together (no inter-arrival time). We also assume each flow has an execution history, so HFMS has a good estimate of its execution time at MPL 1. Also, for each experiment, an engine has a fixed, static MPL, i.e., the number of concurrently executing flows. For each experimental scenario described below, we first run a baseline (control) experiment in which HFMS dispatches flows to engines while maintaining the MPL. But, it takes no other action. For example, if MPL is 3 for one engine, then HFMS will throttle submissions, so that at most three flows are executing on the engine at any time. Then, a series of experiments is run where HFMS processes the same batch at the same MPL but using a flow management policy.

A. Methodology

Our experimental testbed comprises two parallel execution engines. One is an 8 node, parallel, relational database engine (pdb) tuned for business intelligence queries. The second is a 64 node Hadoop 0.20.2 instance (mr). The database nodes each has 8 cores, 48GB memory, and 6 directly attached disk drives, and the nodes reside on the same rack with a private LAN. Each Hadoop node has 6 cores, 24GB memory, and 4 directly attached disk drives, and the nodes reside on 2 racks and communicate on a private LAN. HFMS executes on the same rack as the database engine. HFMS, pdb, and mr share a common LAN for inter-engine communication. Note that an engine comparison is not among our goals here. Although the engine configuration is an input to our system, which then operates according to the environment given, our choice of using a larger cluster for the mr engine relates to the common practice where typically mr follows the scale-out paradigm.

We use three workloads in our study. The first workload comprises flows from the industry standard benchmark, TPC-DS. These flows were expressed as single, nested SQL select queries. We loaded pdb with a TPC-DS dataset at scale factor 300. The fact and large dimension tables were partitioned across all database nodes. For performance comparisons, we created batches of 930 flows, where each batch contained 10 instances of 93 different TPC-DS queries. (We used only 93 of the 100 possible TPC-DS benchmark queries.) The flows were randomly ordered in each batch. At MPL 1 (i.e., running the queries serially), a TPC-DS batch of 930 flows takes approximately 231 minutes to complete. The response time distribution of the TPC-DS flows is shown in Figure 5 (note log scale). As can be seen, most flows finish in under 10 seconds and about 78 percent finish in under 1 minute.

The second workload is a set of 12 ETL flows. These flows are implemented in Apache Pig and run over varying data set sizes ranging from 1 million to 1 billion rows. There are two types of flows. One type models a data cleaning task (including surrogate key generation) and consist of four flows. A second type models a log file analysis task and there are eight of these flows. At MPL 1, a batch of these 12 flows takes approximately 20 minutes to complete on the Hadoop engine, mr (Figure 6).

The third workload is a marketing campaign analysis flow that we used in a previous study of hybrid flows [1]. It consists of 16 flows running over varying data set sizes ranging from 1 million rows to 1 billion rows. They are implemented in SQL and, at MPL 1, a batch of these 16 flows takes approximately 38 minutes to complete on the database engine, pdb (Figure 7).

To investigate runtime adaptation of flows, two sets of experiments were performed. The first set of experiments looked at decomposition of the TPC-DS flows. Recall that these flows were expressed as single, nested SQL queries. This should help the database query optimizer since it sees the full computation in one expression. However, long-running TPC-DS flows may consume a large fraction of pdb resources, starving other flows and limiting throughput. Thus, we investigated a HFMS policy to decompose long-running flows. These flows were re-optimized from a single, nested SQL statement into blocked SQL; i.e., a sequence of sub-flows where the results of each sub-flow were stored into tables that were read by the subsequent sub-flows. The sub-flows were run sequentially in separate sessions. The run times of the decomposed flows

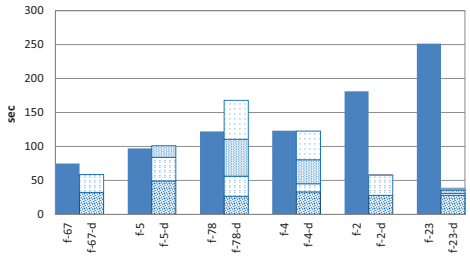
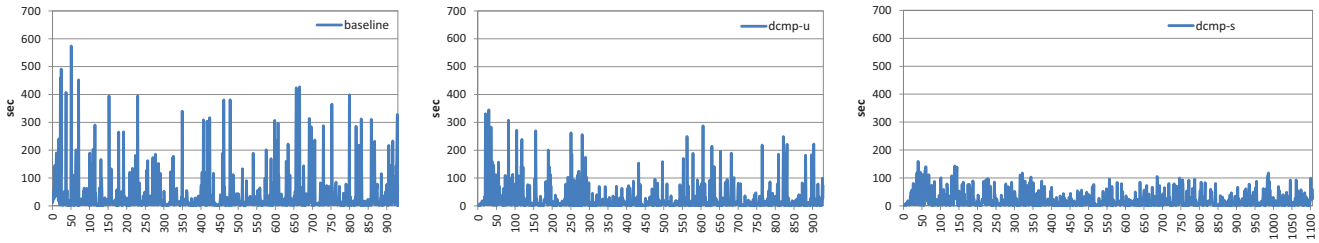


Fig. 8. TPC-DS flow time vs. decomposed flow time



(a) Baseline (b) Flow Decomposition (user) (c) Flow Decomposition (system)

algorithm	♯flows	exec (s)	max (s)	avg (s)	min (s)	♯long	♯short	integral
baseline	930	1:11:10	573.98	35.47	0.42	62	483	16476.0
decomposition (u)*	930	1:03:04	344.82	23.01	0.40	32	583	10689.5
decomposition (s)*	1106	>>	158.87	19.33	0.33	1	641	>>

*The numbers shown here refer to aggregated results obtained from both engines.

(d) Statistics

Fig. 9. TPC-DS flow times, MPL 8, θ infinity for (a) and 60 for (b) and (c)

and the full flows at MPL 1 are compared in Figure 8. As can be seen, in some cases the decomposed flows take longer than the original flows, but sometimes they are faster. This was surprising but, in retrospect, database optimizers are not perfect. Finding the best SQL for a query can take some experimentation which, for a large workload, can be a tedious, time-consuming task. Tools like HFMS can make this much easier by automatically generating and executing different plans for a flow.

The second set of experiments looked at flow shipping, i.e., moving a flow to another engine. Initially, the ETL workload was optimized for Apache Pig on mr and the campaign workload was optimized for SQL on pdb. In this set of experiments, HFMS uses a policy that, when advantageous, redirects a flow to an alternative engine with a new implementation. In these experiments, the required data was preloaded onto all engines, and thus, we did not take into account data shipping costs. The execution times for the ETL flows are compared in Figure 6. It shows that the ETL flows have better performance on pdb for small tasks but, for longer tasks, mr does better than pdb. The execution times of the campaign flows are compared in Figure 7. As with the ETL flows, we see that the campaign flows perform better on pdb for small datasets, but for larger datasets, mr scales better than pdb.

B. Flow decomposition experiments

For these experiments we created six workloads of 930 flows, each containing 10 instances of 93 TPC-DS queries in random order. These flows execute only on pdb. To determine a good policy for decomposition, we needed to find values for two parameters, MPL and a decomposition threshold, θ (see Section VI-C.1). The higher the MPL, the more load on pdb. θ determines when to decompose a flow. If the expected run-time of a flow exceeds θ, the flow is decomposed into sub-flows. We ran all workload batches with MPL at 1, 2, 4, 8, 16 and θ at 20, 60, 100, and infinity seconds (i.e., no decomposition). A representative set of results is shown in Figure 9 for MPL at 8 and θ at 60. These settings worked well for the configuration and workloads tested.

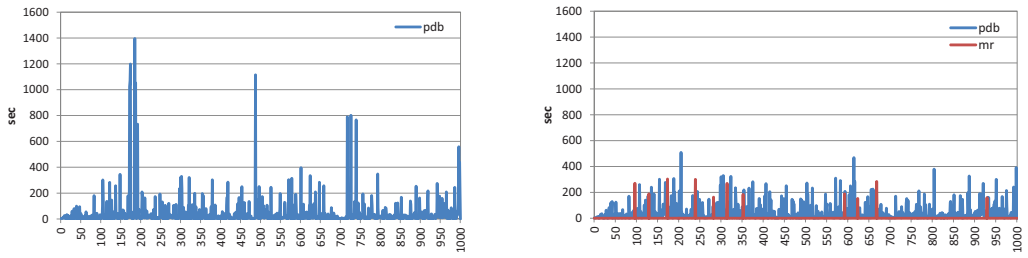
Decomposition reduces the spikes in response time. This is confirmed by the statistics shown in Figure 9(d), which compare baseline performance with decomposition. The integral statistic is the area under the response time histograms which we use as an approximation of overall work. The second row, $dcmp{-}u$, shows the cumulative statistics for a decomposed flow (summed times of sub-flows) as a user perceives it, while the third row, $dcmp{-}s$, computes sub-flows independently, as the system process them. We see that, with decomposition, the number of short-running flows (under 10 seconds) actually increases compared to baseline. Additionally, the impact on the response time of long-running flows (above 150 seconds) is not significant; in fact, the absolute number of the long-running flows decreases. At the same time, decomposition improves the average response time for each flow, makes the system more stable (see also Figure 9(c)), and frees resources to be used otherwise. Figure 9 shows results for one workload consisting of 930 flows (or, for the decomposition, 1106 total flows). However, all workloads have similar results.

Our experiments show that decomposition also improves the overall execution time by some margin (typically, around 10%-15%). Note that this is a welcomed side-effect of our approach, since improving system robustness and avoiding resource contention essentially comes without countermeasures. An additional finding, not shown in Figure 9, is that by reducing the amount of long-running flows, increasing the number of short-running flows, and improving the average execution time of all flows, we enable the system to run effectively at a higher MPL, which in turn, has a positive impact in total response time.

C. Flow shipping experiments

This set of experiments shows the impact of flow shipping on workload performance. Two directions were separately studied, shipping pdb flows to mr and shipping mr flows to pdb. Flow decomposition was not used in order to isolate the impact of flow shipping.

Flows from pdb to mr. This experiment moves work from a heavily loaded engine to a lightly loaded one. Our goal was



(a) Baseline

(b) Flow Shipping

algorithm	♯flows	exec (s)	max (s)	avg (s)	min (s)	♯long	♯short	integral
baseline	1000	1:28:57	1395.14	41.95	0.42	23	283	20973.5
flow shipping*	1000	1:15:01	507.94	40.05	0.39	12	257	20023

*The numbers shown here refer to aggregated results obtained from both engines.

(c) Statistics

Fig. 10. Baseline and flow shipping from pdb to mr for a workload consisting of TPC-DS and campaign flows

to understand the best case for flow shipping, hence, all flows were optimized for pdb. For this experiment, a workload of 1000 flows was created comprising 861 TPC-DS flows and 139 campaign flows, specifically 128 instances of the shorter flows (the 8 leftmost flows in Figure 7) and 11 instances of the longest flows (the 4 rightmost flows in Figure 7), all randomly chosen. The shipping policy moved longer flows from pdb to mr when mr could process them significantly faster. Specifically, the policy moves flows whose pdb cost is greater than x and whose pdb cost is less than its mr cost times ψ; i.e., $x<cost_{pdb}$ *and* $cost_{pdb}<cost_{mr} * \psi$. We tested different values for x and ψ, and we tuned the ratio ψ at 1, favored shipping flows whose expected cost was more than 60 seconds.

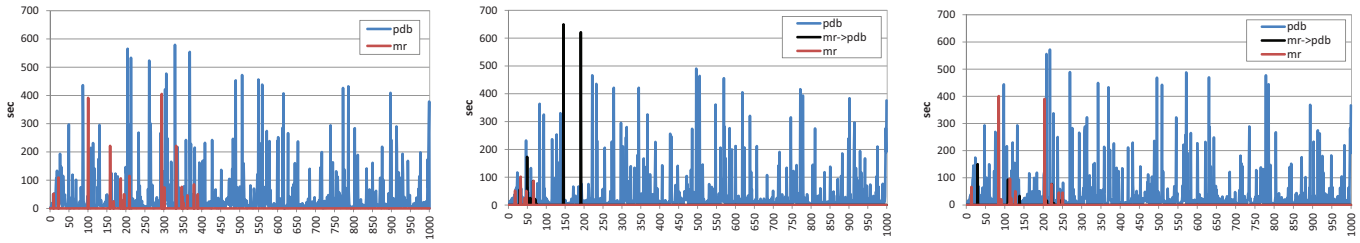
Five random, batch workloads were generated. For each batch workload, we ran a baseline experiment in which all flows were executed on pdb. Then, we ran the same batch workloads with the flow shipping policy in HFMS. In this case, the long campaign flows picked by our policy were redirected to mr. All runs were conducted at MPL 8 for pdb and MPL 1 for mr. Representative results for the baseline are shown in Figure 10(a) and for flow shipping in Figure 10(b). Results for the other workloads were similar.

We had expected flow shipping to reduce execution time of the workload by off-loading work to mr and also to improve average flow response time by reducing contention on pdb. The baseline results show several spikes where the long campaign flows ran longer than 600 seconds. When flow shipping was used, those spikes disappear and only a few flows run over 300 seconds. The statistics, shown in Figure 10(c) tell a more interesting story. The flow shipping workloads do complete faster than the corresponding baselines, about 14 minutes faster (an average of 15%-20% for all workloads tested). However, compared to the baseline, most TPC-DS flows ran slower on pdb under flow shipping; e.g., as Figure 10(c) shows with flow shipping we have a reduced number of short-running flows (under 5 seconds). Only the longest flows (above 300 seconds) ran faster and in fact, with flow shipping the number of long-running flows almost halved. In reviewing the logs, we found that moving the long campaign flows to mr effectively increased contention on pdb, because it increased the chance that two moderately long TPC-DS flows would run concurrently. This increased the chances that yet a third somewhat long flow would arrive. In other words, a *convoy* effect occurs. We believe this does not occur for the very long flows, because they are so large they prevent other flows from making any progress; i.e., no new flows is dispatched while they execute.

Regardless of the policy used for flow shipping, the scheduler also accounts for the load of the other engine(s) too. The extra load added to the mr engine for this experiment ran approximately for half the time pdb needed to complete its workload (around 40 minutes for the numbers reported in Figure 10). Shipping more or fewer workloads on mr depends on how the scheduler tunes the shipping policy based on the service level agreements, costs calculation, and waiting time prediction. Our intention here is not to detail this process, but rather to show the feasibility of flow shipping to exploit idle engine resources and to speed-up flows awaiting execution on a busy engine.

Flows from mr to pdb. This experiment moves work from a moderately loaded engine to a heavily loaded one. An administrator might do this if the moderately load machine is costly to use, e.g., a server on a public cloud. Our goal is to understand the impact on the heavily loaded engine. For this experiment, a workload of 1000 flows was created comprising 980 TPC-DS and 20 ETL flows, again, all randomly chosen. The ETL flows comprise one instance of each of the 4 data cleaning tasks and two instances of each the 8 log file analysis tasks shown in Figure 6. Initially, the TPC-DS flows were designed and optimized for execution on pdb and the ETL flows on mr.

Five random, batch workloads were generated. For each batch workload, we ran a baseline experiment in which all flows were executed on the engine chosen by the optimizer. The results of the baseline experiment are shown in Figure 11(a). Next, we ran the batch workloads using a flow shipping policy that moved flows from mr to pdb when its pdb



(a) Baseline (b) Flow Shipping (ψ=1) (c) Flow Shipping (ψ=3)

algorithm	♯flows	exec (s)	max (s)	avg (s)	min (s)	♯long	♯short	integral
baseline	1000	1:20:20	579.35	40.07	0	21	653	20054
flow shipping* (ψ=1)	1000	1:16:00	571.57	37.14	0.52	19	663	18572.5
flow shipping* (ψ=3)	1000	1:11:21	649.08	33.99	0.39	21	708	16991.5

*The numbers shown here refer to aggregated results obtained from both engines.

(d) Statistics

Fig. 11. Baseline and flow shipping from mr to pdb for a workload consisting of TPC-DS and ETL flows

cost times ψ is less than its mr cost, i.e., $cost_{pdb} * \psi < cost_{mr}$. The move threshold was set to zero to off-load as much work as possible from mr. In this experiment, we wanted to study flow shipping from mr to pdb in isolation and thus, we tweaked the policy to prevent transferring flows in the opposite direction, i.e. pdb to mr.

As before, we tested various ψ values. For larger ψ values (e.g., greater or equal to 3), flow shipping was more conservative. For smaller ψ values (e.g., between 1 and 2), we observe more aggressive movement. Figures 11(b) and 11(c) illustrate our findings for a single workload and for ψ at 1 and 3, respectively. Observe, both the positive and negative effects of each ψ value.

For more aggressive shipping, as in Figure 11(b), most of the costly ETL flows have been transferred to pdb resulting in an additional overhead on that engine. Note the two large ETL flows running concurrently on pdb (flows 145 and 191). However, while the mr engine was released from most of its duty (around 80% load reduction), Figure 11(d) shows pdb was not much affected. In fact, most of the statistical measures were slightly improved. For the conservative flow shipping, as in Figure 11(c), only the very expensive ETL flows were shipped to pdb. This action removes some significant load from mr, adds some extra load to pdb, and the systems are more balanced than before. For the case shown in Figure 11(c), we managed to reduce the mr load by almost 40%, and although this load was transferred in pdb, the careful scheduling managed to achieve this in smaller total execution time (about 9 minutes). Results for the other workloads were similar.

Use case summary. To abstract ourselves from overwhelming details and focus on the gist of the approach, we described here a few rather simplistic policies. More realistic policies, which would be more likely used in practice, would consider the current system load, anticipated load, the service level objectives and penalties, etc. Our goal was to study the impact of the actions. As an overall comment on the use case, it is worth noting that our scheduling strategies do not aim at replacing the optimizer. In fact, the preferable action would always be to follow the optimizer plan. However, since an HFMS flow may be executed at different times and not necessarily immediately after optimization, we need to account for the most current status of the execution environments before execution. It is exactly then, when these techniques kick in to groom the flows. And typically, as we see in the experiments too, this extra touch does not affect the majority of the flows in a workload. Optimizing just before execution is not always feasible, since for supporting future reservations and to avoid over-provisioning we need to have a good plan at hand.

VIII. Related Work

Many systems have the ability to generate a single type of executable code from a logical graph. However, no systems are able to generate code for a variety of targets. Some systems can generate code within a particular family of targets. For example, some database query systems can generate SQL code for a variety of database systems (e.g., PostgreSQL, MariaDB). Some systems have the ability to generate executable code for more than one execution engine. Many ETL systems (e.g., PDI, Informatica, Ab Initio) can generate code to, for example, filter and extract data from database systems and then import and process that data into the native ETL engine. However, these systems are inflexible in that they generate identical executable code for a given graph. Our system can generate a variety of different executable forms from a single graph, e.g., given a logical graph, we might create one executable form that filters a data set in a database system, while another form might filter that data set using a scripting engine.

Research on federated database systems considered query optimization across multiple execution engines. But, this work was limited to traditional relational query operators and to performance as the only objective; for example, see query optimization in Garlic [12], Multibase [13], and Pegasus [14]. Same limitations have the works focusing on query optimization for traditional database systems, which focus solely on performance and consider a subset of operators typically encountered in our case [15], [16], [17].

Several high-level languages can be translated to execute on a map-reduce engine (e.g., JAQL [18], Hive [19], Pig [20]).

These languages offer opportunities for optimization. Other research efforts have generalized the Map-Reduce execution engine to process a large class of parallel-distributed data flows (e.g. Asterix [21], CIEL [22], Dryad [23], Nephele [24]). Such systems typically have higher-level languages that can be optimized and compiled to execute on the parallel execution engine (for example, see PACTs in Nephele, DryadLINQ in Dryad, SCOPE, Skywriting in CEIL). None of these projects addresses data flows that span different execution engines though. HadoopDB is one example of a hybrid system in which queries span execution engines [25]. HadoopDB stores persistent data in multiple PostgreSQL engines. It supports SQL-like queries that retrieve data from PostgreSQL and process the data over Hadoop. However, it is not designed as a general framework over multiple engines. Another approach, namely SQL/MapReduce has been successfully implemented in Teradata Aster [26]. Although in that framework one may write a SQL query that spans the map-reduce and database engines, to the best of our knowledge, it is not possible (yet) to optimize the map-reduce tasks along with the database operators. And in any case, it is not designed for producing different flow variations.

Workload management tools are offered for most commercial database management systems. But, they do not support workloads over other engines. There has been research on hybrid workloads (e.g., combining OLTP and analytic queries), but the assumption is that one engine executes all workloads [27]. Load balancing tools for different processing engines exist. But, their capabilities are limited to job redirection.

IX. Conclusions

Enterprises are finding that their business intelligence needs cannot be fully met by a centralized data warehouse supplemented by data marts. They are now deploying a variety of processing engines that are tailored for specific tasks and data objects. A common example is the use of map-reduce engines for analysis of log files and other unstructured data. Thus, answering a business question may involve operations that span many storage repositories and execution engines. Conceptually, this is a single analytic data flow. But, the current approach to implementing such *hybrid* flows is stepwise, utilizing a sub-flow of operations on one engine and transferring the intermediate results to the next engine. Alternatively, some engines offer specialized connectors to other engines. In either case, engines are not treated as peers and the individual engines are unaware that they are working on part of a larger flow.

In this paper, we introduced HFMS, a platform for creating, optimizing, and executing hybrid flows. HFMS treats hybrid flows as unified, single entities. It exists as a layer above independent execution engines. An important feature is its ability to optimize flows for multiple objectives and for different execution engines. It also provides workload management for hybrid flows that leverages the optimizer to create alternative executable plans for flows depending on the current system state. To motivate the benefit of HFMS, we presented a use case of workloads running on two example engines, a database and map-reduce engines, and show how management actions like decomposition and flow shipping can improve quality of service.

We do not claim that HFMS does a better job at optimization, flow processing or workload management than any particular execution engine. Our point is that the current, adhoc approach cannot scale. The task of building and maintaining flows that span multiple engines and administrative domains demands automation. Although there are many open issues, e.g., transactions and consistency, error handling, cloud, security, we believe HFMS is a step in the right direction.

References

[1] A. Simitsis et al., "Optimizing analytic data flows for multiple execution engines," in SIGMOD, 2012, pp. 829–840.

[2] Pentaho, "Pentaho pdi. url: http://kettle.pentaho.com/," 2012.

[3] A. Simitsis and K. Wilkinson, "Revisiting ETL benchmarking: The case for hybrid flows," in TPC-TC, 2012.

[4] C. Gupta et al., "PQR: Predicting query execution times for autonomous workload management," in ICAC, 2008, pp. 13–22.

[5] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems," in PODS, 1998.

[6] P. G. Selinger et al., "Access path selection in a relational database management system," in SIGMOD, 1979, pp. 23–34.

[7] G. Graefe and W. J. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in ICDE, 1993, pp. 209–218.

[8] A. Simitsis et al., "Optimizing ETL workflows for fault-tolerance," in ICDE, 2010, pp. 385–396.

[9] H. T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," J. ACM, vol. 22, no. 4, 1975.

[10] E. Iori et al., "CloudAlloc: a monitoring and reservation system for compute clusters," in SIGMOD, 2012, pp. 721–724.

[11] B. Schroeder et al., "How to determine a good multi-programming level for external scheduling," in ICDE, 2006, p. 60.

[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] W.-S. Han et al., "Parallelizing query optimization," PVLDB, vol. 1, no. 1, pp. 188–200, 2008.

[16] G. M. Lohman et al., "Query processing in R*," in Query Processing in Database Systems. Springer, 1985, pp. 31–47.

[17] T. K. Sellis, "Multiple-query optimization," TODS, vol. 13, no. 1, pp. 23–52, 1988.

[18] K. S. Beyer et al., "Jaql: A scripting language for large scale semistructured data analysis," PVLDB, vol. 4, no. 12, pp. 1272–1283, 2011.

[19] A. Thusoo et al., "Hive - a petabyte scale data warehouse using Hadoop," in ICDE, 2010, pp. 996–1005.

[20] C. Olston et al., "Pig latin: a not-so-foreign language for data processing," in SIGMOD, 2008, pp. 1099–1110.

[21] A. Behm et al., "Asterix: towards a scalable, semistructured data platform for evolving-world models," Distributed and Parallel Databases, vol. 29, no. 3, pp. 185–216, 2011.

[22] D. G. Murray et al., "CIEL: A universal execution engine for distributed data-flow computing," in USENIX NSDI, 2011.

[23] M. Isard et al., "Dryad: distributed data-parallel programs from sequential building blocks," in EuroSys, 2007, pp. 59–72.

[24] D. Battré et al., "Nephele/PACTs: a programming model and execution framework for web-scale analytical processing," in SoCC, 2010.

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

[26] 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.

[27] S. Krompass et al., "A testbed for managing dynamic mixed workloads," PVLDB, vol. 2, no. 2, pp. 1562–1565, 2009.