Revisiting ETL Benchmarking: The Case for Hybrid Flows

Alkis Simitsis and Kevin Wilkinson

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

Abstract. Modern business intelligence systems integrate a variety of data sources using multiple data execution engines. A common example is the use of Hadoop to analyze unstructured text and merging the results with relational database queries over a data warehouse. These analytic data flows are generalizations of ETL flows. We refer to multi-engine data flows as hybrid flows. In this paper, we present our benchmark infrastructure for hybrid flows and illustrate its use with an example hybrid flow. We then present a collection of parameters to describe hybrid flows. Such parameters are needed to define and run a hybrid flows benchmark. An inherent difficulty in benchmarking ETL flows is the diversity of operators offered by ETL engines. However, a commonality for all engines is extract and load operations, operations which rely on data and function shipping. We propose that by focusing on these two operations for hybrid flows, it may be feasible to revisit the ETL benchmark effort and thus, enable comparison of flows for modern business intelligence applications. We believe our framework may be a useful step toward an industry standard benchmark for ETL flows.

1 The Emergence of Hybrid Flows

The practice of business intelligence is evolving. In the past, the focus of effort was on ETL to populate a data warehouse. ETL data flows extract data from a set of operational sources, cleanse and transform that data, and finally, load it into the warehouse. Although there are common flow paradigms, there are no industry standard languages or models for expressing ETL flows. Consequently, a variety of techniques are used to design and implement the flows; e.g., custom programs and scripts, SQL for the entire flow, the use of an ETL engine. Flow designers must choose the most appropriate implementation for a given set of objectives. Based on their level of expertise, their choice may be sub-optimal. The industry lacks good tools such as standardized benchmarks and flow optimizers to enable designers to compare flows and improve their performance.

The success of industry standard benchmarks such as the TPC suites led to hope that similar benchmarks could be developed for ETL flows. An exploratory committee was formed, but so far no results are publicly available. The various ETL engines offer a diverse set of features and operators, so it is difficult to choose a common set for a meaningful comparison. However, the need for an

R. Nambiar and M. Poess (Eds.): TPCTC 2012, LNCS 7755, pp. 75–91, 2013.

ETL benchmark, as envisioned, is less relevant now because the demands on business intelligence have changed. The traditional BI architecture used periodic, batch ETL flows that produced a relatively static, historical view of an enterprise on a centralized, back-end server. Enterprises now require a dynamic, real-time views of operations and processes. To enable these views, flows must integrate numerous, dispersed data sources in a variety of data formats. These flows may utilize multiple processing engines, some general-purpose and some special-purpose for a particular type of data. We refer to these multi-engine flows as *hybrid flows*. For hybrid flows, there is no single, most appropriate engine for the entire flow. Instead, the designer must choose how to partition the flow into sub-fragments that each run on different engines.

For an example of a hybrid flow, consider a hypothetical consumer product company that desires real-time feedback on new products. To do this, one flow might load product commentary from sources like Twitter and Facebook into a map-reduce cluster (e.g., Hadoop) and use text analytics to compute customer sentiments. Separately, a second flow might aggregate retail sales data from an operational data store. The results of these two flows would then be joined to correlate sales to product sentiment, and thus, evaluate product launches.

Designing and implementing a correct hybrid flow is difficult, because such flows involve many computing systems and data sources. Optimizing a hybrid flow is an even more difficult and challenging task. For example, there may be overlapping functionality among the execution engines, which presents a choice of engines for some operators. Also, the engine with the fastest operator implementations may not be the best choice since the design must consider the cost of shipping input data to that engine. On the other hand, some operators in the flow may run on multiple engines while other operators may require a specialized engine. Some operators have multiple implementations with different characteristics (e.g., sorted output, blocking or pipelined execution). Some engines provide fault-tolerance. Consequently, the number of alternative designs grows exponentially. In our view, the role of the (human) flow designer is to create a correct, logical hybrid flow. An optimization tool should then be used to find an alternative design (i.e., a partitioning of the flow such that different flow fragments may run on different engines) that meets the optimization objectives. Our research group has been developing the QoX optimizer to do this.

The cost of transferring large datasets is a critical factor in choosing the best partitioning for a hybrid flow. Hence, a key challenge for the optimizer is obtaining good cost estimates for *data* and *function shipping*, as we detail in Section 3. Poor estimation risks either pruning good designs or choosing designs with bad performance. The QoX optimizer derives its estimates from a set of microbenchmarks for data shipping and for function shipping. For a given pair of repositories, a series of data transfer experiments are run to extract data from the source repository and load it in the target repository. The optimizer can then interpolate over these results to estimate the data transfer time for an operator in a given flow. The microbenchmarks are effectively extract and load operations. As such, they could form the basis for an ETL benchmark suite.

In the next section, we present an example analytic, hybrid data flow and we discuss it through our optimizer. We show alternative hybrid designs, each with a different partitioning of the flow into subflows. We compare the execution times for the alternative designs and discuss performance factors. Section 3 discusses the optimizer microbenchmarks themselves including metrics for hybrid flows and infrastructure for data collection. Section 4 presents a collection of parameters to describe hybrid flows. Parameters like these would be required in any general framework to benchmark hybrid flows. Section 5 reviews related work and Section 6 concludes the paper.

2 QoX Optimizer for Hybrid Flows

The input to the QoX optimizer is a logically correct data flow, expressed as a directed graph of operators and source and target data stores, and a set of objectives. The optimizer generates alternative, functionally equivalent, flow graphs using graph transitions such as operator swap, flow parallelization, insert recovery point. The execution cost of each alternative is estimated and compared to the objectives. Heuristic search is used to prune the search space.

For each operator in the flow graph, the optimizer identifies all available implementations on all the execution engines. For example, filtering operators and scalar aggregation operators might be offered on all execution engines while some specialized operators, such as k-means clustering, might be available on just a single engine. The source and target datasets may be initially bound to specific repositories, e.g., HDFS (Hadoop file system), UFS (Unix file system), SQL engine. However, the optimizer will consider shipping the datasets to other repositories to improve the flow.

For a given a flow graph the optimizer must assign operators and datasets to execution engines. It performs an initial assignment using first-fit starting with the source datasets and traversing the flow graph. It then uses two graph transitions, data shipping and function shipping, to generate alternative feasible assignments. Function shipping reassigns the execution of some operator from one execution engine to another engine that supports the operator. Data shipping copies a dataset from one data repository to another. Note that function shipping may induce data shipping if the data is not local to the engine and that must be included in the cost of function shipping.

As an example of function shipping, consider a binary operator assigned to one execution engine, but with input datasets from two other engines. Moving the binary operator to execute on the engine with the largest input will minimize data movement and so, this may be a better plan. For data shipping, a common example is when an ETL engine extracts data from an SQL engine. As another example, suppose an operator can only read from a text file. If the operator input happens to be stored in a relational database, the optimizer must insert a data shipping operator to copy the table to the file system.

Hence, given a hybrid flow, the QoX optimizer partitions it into sub-flows that each run on separate engines. There are many possible cut points for partitioning

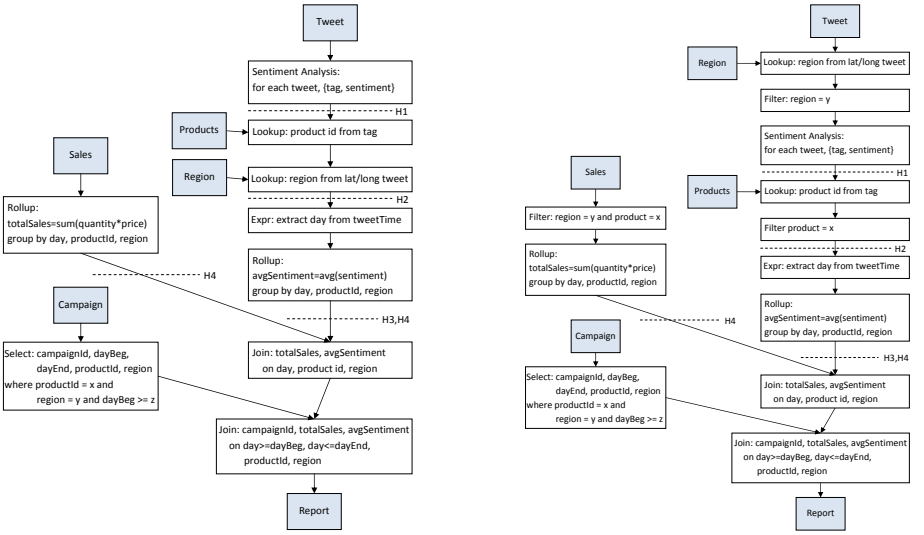

Fig. 1. Example flow combining structured and unstructured information (left) and an optimized variant of the flow (right) with each showing possible flow partition choices

a flow. The function shipping and data shipping transitions enable the optimizer to consider all feasible partitionings. The design with the lowest estimated cost relative to the objectives is chosen. The final graph is a collection of sub-flows, each assigned to execute on a single execution engine, and with data shipping operators used to connect the sub-flows.

Example Flow. The left side of Figure 1 shows a real-world, analytic flow that combines free-form text data with structured, historical data to populate a dynamic report on a dashboard. The report joins sales data for a product marketing campaign with sentiments about that product gleaned from tweets crawled from the Web. The report lists *total sales and average sentiment for each day of the campaign*. Campaigns promote a specific product and are targeted at non-overlapping, geographical regions. The sentiment analysis of a tweet yields a single metric, e.g., like or dislike the product over a range of -5 to +5.

Our example flow starts with text analysis that computes a sentiment value for a product mentioned in a tweet. Then, two lookup operators are performed, one that maps product references in the tweet (e.g., ENVY Spectre, TopShot LaserJet 3) to a specific product identifier and a second that maps latitude and longitude of the tweet to a geographical region. Then, the tweet timestamp is converted to a date and the sentiment values are averaged over each region, product, and date. On a parallel path, the sales data is rolled up to compute total sales of each product for each region and day (assume the sales table includes the region of the sale). Next, the rollups for sales and sentiment are joined and finally the specific campaign of interest is selected and used to filter the result. The right side of Figure 1 shows the optimized flow generated by the

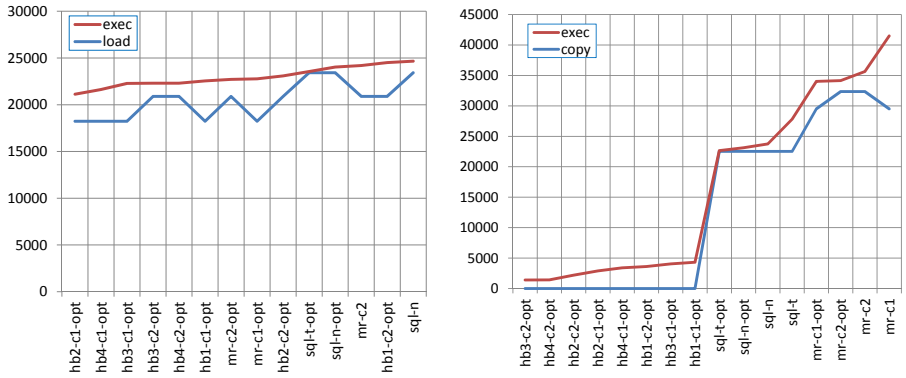

Fig. 2. Load (left) and Copy (right) times for 10G rows

QoX optimizer. The details of the flow restructuring are not described here (for details, see [11]) since our focus is on the flow partitioning.

In our example, we assume a system configuration comprising a map-reduce engine, MR, and a parallel database engine, pDB, each engine running on a separate set of nodes with no shared storage and all nodes connected with a single LAN. Each dataset is bound to a repository. Tweets are stored on the distributed file system of MR and the remaining four datasets are stored as relational tables distributed across all nodes of pDB. The sentiment analysis operator is only supported on MR while all other operators are supported on both engines.

We discuss four alternative assignments of sub-flows to execution engines for both flows of Figure 1. The first multi-engine flow (hybrid flow hb1) executes the sub-flow up to the sentiment analysis operator on MR and the remaining operators on pDB. This cut point is denoted by $H1$ in Figure 1. The second multi-engine flow (hb2) adds product lookup to the previous flow to be executed on MR. This cut point is denoted by $H2$. The third multi-engine flow (hb3) performs the sentiment rollup operator on MR and the cut point is denoted by $H3$. The fourth hybrid flow (hb4) performs two sub-flows in parallel, specifically, the MR rollup sub-flow and the rollup of sales data. Next, these two rollups are joined in pDB and then, joined with campaign data. This cut point is denoted by $H4$ in Figure 1.

The relative merits of the various partitionings depend on the dataset sizes as well as our assumptions about the initial bindings of datasets to repositories. Figure 2 shows the effect of load and copy times (in sec) in a 10G rows dataset for various flow configurations on two clusters, a smaller c1 (16 nodes) and a larger c2 (32 nodes) clusters. (In this experiment, 10 billion rows of tweet data occupy 1.22TB disk space, while for the other datasets the same amount of rows needs around 270GB of disk storage.) Load refers to the case where we load data from the file system to an engine; here from the filesystem to the MR and pDB according to the flow. Copy refers to the data shipping from one engine to another; here from MR to pDB. In the stacked lines of Figure 2, the execution times (exec) add up to the copy and load times, in order to get the total

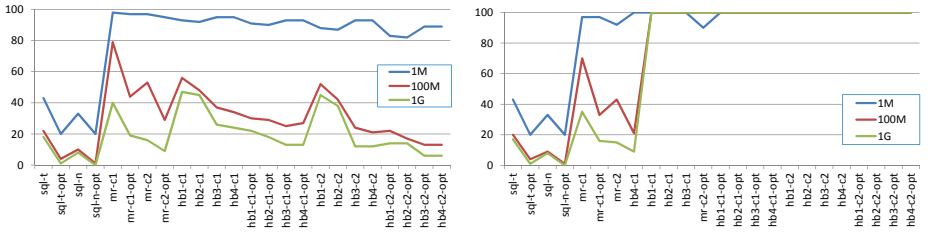

Fig. 3. Effect (%) of load (left) and copy (right) with varying sizes

processing time for a flow. The graphs show that the copy and load dominate the total times. However, we observe that single engine policies (mr for MR, sql for pDB) do not give the best results in both cases, while the hybrid flows (hb-x) perform much better. In particular, the sql-x cases (both the unoptimized sql-n, sql-t and the optimized sql-n-opt, sql-t-opt) although they perform really well in terms of execution time, their total performance suffers from the load/copy times, and thus, in total, these are not good solutions.

Similar observations may be made by looking how each flow variant performs for different input sizes. Figure 3 shows different flow configurations (both single engine and hybrid flows) for varying sizes of 1M, 100M, and 1G rows. These lines shows percentages: values below the lines show the percentage of load and copy times, and values above the lines show the percentage of execution times for the different data sizes. We observe that the negative effect of load and copy times in the total performance decreases with the data size and this in general, is in favor of the hybrid flows.

Both these experiments show the significance of data and function shipping, especially as the data size increases.

3 Metrics and Benchmarks for Data and Function Shipping

3.1 Benchmark Design for Data Shipping

We now formulate the problem of estimating data shipping costs for a computing system configuration. A computing system comprises a number of nodes and a set of execution engines. Some engines execute in parallel on a subset of nodes whereas others may be single node engines. A computing system has one or more storage repositories. As with execution engines, a repository may be local to a single node or be distributed, storing its data objects across a set of nodes. A repository provides a namespace to identify objects and, at a minimum, operations to create, destroy, read, and write objects. To simplify the discussion we assume a repository supports a single data representation/format (e.g., table, key-value pairs, XML). In practice, some engines support multiple data formats, but we consider them here as logically distinct repositories.

Table 1. Data Paths matrix

src/tgt	$repo_1$	$repo_2$	...	$repo_S$	null
$repo_1$	-	p_{12}	...	p_{1S}	x_1
$repo_2$	p_{21a},p_{21b}	-	...	-	x_2
...	...	...	...	...	...
$repo_S$	-	p_{S2}	...	-	x_S
null	l_1	l_2	...	l_S	-

For each storage repository, we need cost estimates for shipping data to other repositories. We also need costs for loading data to the repository and for extracting data from the repository (e.g., to/from an application). We use the generic term *path* to refer to a direct data transfer method from one repository to another. Path also refers to methods to extract from or load to a repository. Each repository has its own storage format so a path handles data reformatting/transformation as needed.

Assume there are s possible repositories. Then, we can represent the data shipping costs as an $s \times s$ matrix where each cell, p_{ij}, represents a path for data movement from a source repository i to a target repository j (see an example Data Paths matrix in Table 1). Each path has an associated method (executable program) to perform the data transfer. Note that there may be multiple paths from a source repository to a target, e.g., most SQL engines can store data to a text file either by using a "select into file" statement or by using an export tool. A null source or target signifies an unconstrained path, representing the highest possible data load rate or data extract rate; e.g., use of a high-speed, artificial data generator as the source in loading a target repository. Note that the matrix is not symmetric, i.e., a path in one direction does not imply an inverse path and, if there is one, the cost may differ.

A set of metrics is associated with each non-empty cell in the matrix. To simplify the discussion, we assume a single metric, elapsed time. But depending on the optimization objectives, other metrics may be relevant such as utilization, average throughput, and so on. In addition, each path has an associated set of properties that may be useful to the optimizer, e.g., is blocking or pipelined, output is ordered, is parallelizable, and so on.

The Data Paths matrix defines the feasible direct transfer paths for the optimizer to consider. For each path, the optimizer needs cost formulae to estimate data transfer costs. These are obtained by executing a series of microbenchmarks that exercise a transfer path for varying dataset sizes. The results can be used with a regression algorithm to derive a cost formula or else stored in a data structure for later lookup and interpolation by the optimizer. If there is no direct path between two repositories, the optimizer may consider multi-hop transfer paths by linking direct paths; e.g., in Table 1, to ship data from $repo_S$ to $repo_1$ the optimizer may use path p_{S2} followed by p_{21a} or p_{21b}.

Data shipping costs are not static. Data center infrastructure undergoes periodic change, e.g., software upgrades, replacement of compute racks, introduction and retirement of applications, and so on. Consequently, we must automate the

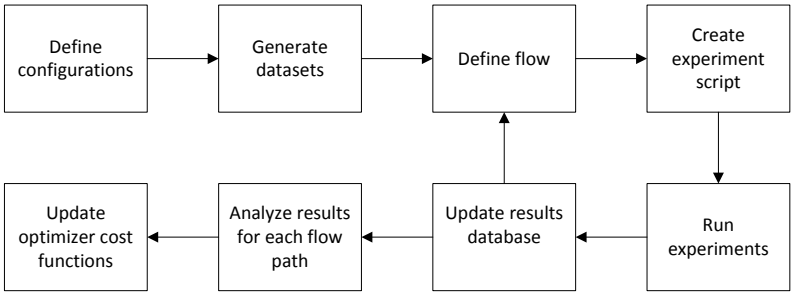

Fig. 4. Steps to create cost function for data transfer path

collection of metrics for data shipping to maintain accurate estimates. To do this, we adapted the technique used by database systems to calibrate query optimizers. When porting to a new platform, database engineers run a series of microbenchmarks to determine the resources required for each operator; e.g., scan a table, do an index lookup, compare two data values, copy a character string. These measurements are used to tune the query optimizer cost estimates for the various database operators.

Our QoX optimizer estimates costs for data transfer paths by following the steps illustrated in Figure 4. At a high level, the process can be summarized as follows. For a given data path, we define a base experiment to transfer data across the path and then run the experiment and measure its performance. We then vary the base experiment, e.g., by scaling the source dataset size, and run those experiments and repeat. Once we have sufficient data points, we derive a cost formula and add it to the QoX optimizer.

At a more detailed level, the initial step is to define the computing system configuration used by a path. A path configuration includes, for both the source and target, the physical nodes, the execution engines on those nodes and the storage repositories. For example, consider a path that copies a distributed file from a map-reduce engine and stores it as a text file on the file system of a single node. The path configuration includes the physical nodes for the map-reduce engine and for the single node, the engines are the map-reduce engine and the operating system of the single node, and the repositories are the distributed file system and the local file system of the node.

The second step is to identify the datasets used in the experiments. Then, we create a metadata description of the flow (see also Figure 5 as we explain below). This comprises an identifier and textual description, links to the source and target datasets, and the path configuration. This metadata is linked with the metrics in the results database to provide provenance. The next step is to define a script or program to execute the flow. At this point, we can now conduct experiments. To reduce random error, we run each experiment a number of times. Metrics are collected in the results database. Once we have sufficient data points, we may create a new flow by altering the flow in any number of ways, e.g., by scaling the datasets, by modifying the node counts or adjusting software

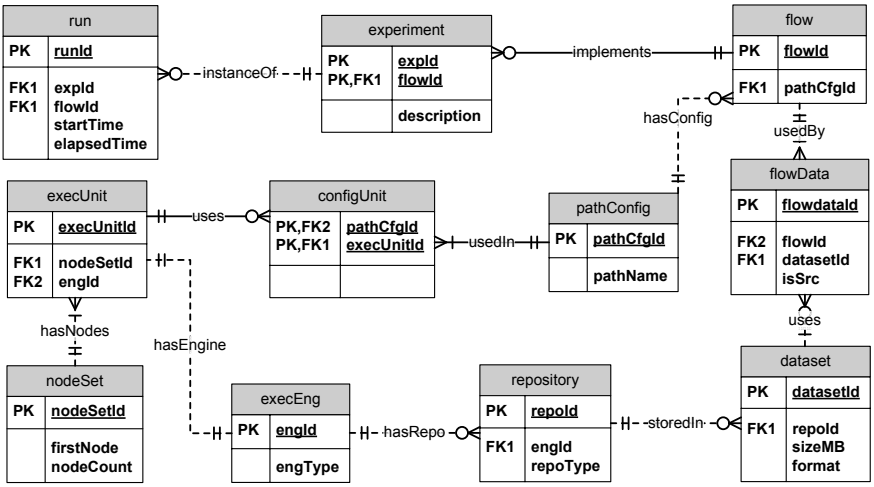

Fig. 5. Benchmark schema

configuration parameters such as replication level or block size. We then run more experiments with the new flow. Eventually, we break out of the loop and derive a cost formula for the path.

A synopsis of the schema used in the results database is shown in Figure 5. Each path is represented by a flow object. A flow has a number of associated experiments (e.g., at different scale factors) and, for each experiment, there are some number of runs. Each flow has a source and target dataset and each dataset is bound to some repository. Additionally, the flow is linked to its configuration that identifies the execution engines and nodes used by the path. The schema shown in Figure 5 is a simplified version of the actual schema used by the QoX optimizer. That schema is designed to support arbitrary hybrid flows, not just single source-target data transfers.

For a given flow, the metrics for a set of experiments can be extracted from the results database and graphed as in Figure 6. This first example (Figure 6, left) shows the time to load a dataset at different scaling factors for three repositories: the distributed file system of a map-reduce engine, a parallel database system, and the local file system for a node. The parallel systems outperform the single node for small datasets, but all systems converge to the same limiting performance for larger datasets.

The second example (Figure 6, right) shows the time to transfer the same datasets from the parallel database system to the distributed file system of the map-reduce engine. There are two transfer paths, a serial path that ships the data through a single node of each engine and a parallel path that ships data concurrently using all nodes of both engines. As can be seen, the serial path shows log-linear scalability and out-performs the parallel path for small datasets. This is because the parallel path has high initial overhead, e.g., it must start processes on each node. However, for large datasets, this overhead is a small fraction of

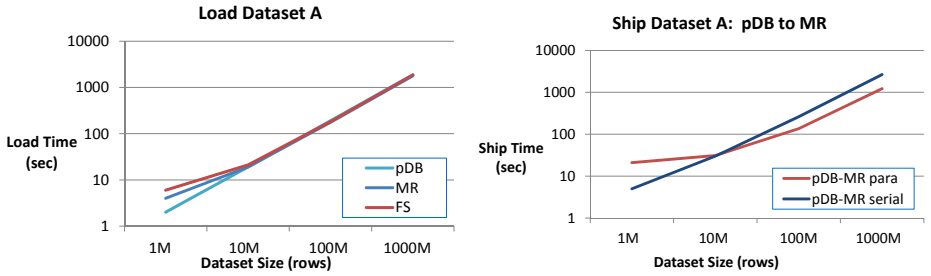

Fig. 6. Load results (left) and two data paths for shipping data from pDB to MR (right)

the total transfer time so the parallel path can leverage the additional resources to outperform the serial path.

3.2 Benchmark Design for Function Shipping

The Qox optimizer generates alternative flow graph designs using both data shipping and function shipping transitions. Section 3.1 describes how we derive cost estimates for data shipping. In this section, we propose a similar technique to estimate the cost of function shipping. For each flow operator f, we associate a set of pairs, $\{m_f\}$, where each pair specifies an implementation of f on an execution engine; e.g., an operator to generate content-based keys using the SHA-1 hashing algorithm on a map-reduce engine.

Assume a flow has an operator f assigned to an engine e_x. The optimizer will consider alternative flows in which f is executed on all other implementations and engines in $\{m_f\}$. If there are p execution engines, we can represent the cost of function shipping by a $p \times p$ matrix (see Table 2) where a cell entry, c_{xi} is the cost of shipping the execution of f from engine e_x to engine e_i. Note that a cell may have multiple entries if the target engine supports multiple implementations for the operator; e.g., a database engine with more than one join method.

Table 2. Function shipping matrix

src/trgt	eng_1	eng_2	...	eng_P
eng_1	c_1	c_{12}	...	c_{1P}
eng_2	c_{21a},c_{21b}	c_2	...	-
...	...	...	...	...
eng_P	-	c_{P2}	...	c_P

In Table 2, *src* is the execution engine with direct access to the storage repository for the input(s)[1] to operator f. The execution engine that actually executes

[1] Here, for the sake of presentation, we assume that all inputs of f refer to the same repository. We assume specialized connectors to connect different repositories. But one may easily generalize our thoughts for hybrid n-ary operators that get their inputs from more than one repository.

src/trgt	eng_1	eng_2
eng_1	1	$c_{12} * 2$
eng_2	c_{21}	2

src/trgt	eng_1	eng_2
eng_1	-	$c_{12} * 4$
eng_2	-	4

src/trgt	eng_1	eng_2
eng_1	-	$\min(c_{12} * 4, c_{12} * 8)$
eng_2	-	$\min(c_{21} * c_{12} * 4, 8)$

Fig. 7. Shipping matrices for functions f, g, and $g(f)$

operator f is *trgt*. The diagonal (*src* same as *trgt*) is the case where the data and operator are on the same engine so c_i is just the operator cost (or null if the engine does not implement the operator). Of course, we may have more than one cost c_i in the diagonal, if more than one implementation is supported on the engine e_i. If *src* differs from *trgt*, then the input data must be shipped to *trgt*. This shipping cost estimate should be added to c_{xi}.

A typical flow contains a sequence of operators, so the optimizer must compute function shipping costs for operator composition. This is accomplished with the function shipping matrix using the distance product matrix computation. In other words, to compose operators f and g, the function shipping matrix for the composition is the distance product matrix multiplication of $\{m_f\}$ and $\{m_g\}$. As an example, suppose the function shipping matrices for f and g over two engines are given by the left and middle tables in Figure 7. Note that eng_1 does not implement g. Their composition $g(f)$ is given by the rightmost table in Figure 7.

In order to calibrate the optimizer, we conduct function shipping experiments similar to the data shipping experiments. For the various functions, we create simple flows and measure the performance over artificial datasets. Then, we scale the experiments and conduct more experiments to gather sufficient data to derive a cost formula. From the cost formula for single operators, the optimizer can compute costs for operator composition.

4 Benchmark Parameters for Hybrid Flows

Data shipping and function shipping are important aspects in the operation of hybrid flows. However, other parameters are of interest too. In this section, we provide a list of parameters and variants that should be considered for designing a benchmark for hybrid flows. We classify them into the following categories: flow, engine, operator, and data related variants.

4.1 Flow Related Variants

We create and measure flows with varying characteristics, as follows.

Flow Size: The number of operators (♯ops) and data stores (♯dst) contained in a flow.

Engine Multiplicity: The number of engines (♯eng) that can be used for the flow execution.

Transition Likehood: A percentage tr% of possible transitions (e.g., swap, factorize, parallelize, function/data shipping) allowed for a flow. This parameter

enables the creation of flows that can be further optimized. This will give us the flexibility to create equivalent variants of the flow produced.

The transition likehood may be further analyzed per transition, i.e., tr% may be read as X%, where:

swapping factor: X = swa
factorization factor: X = fac
distribution factor: X = dis
add-partitioning factor: X = par
data-shipping factor: X = dsh
function-shipping factor: X = fsh
add-replication factor: X = rep
add-recovery-points factor: X = rec

For example, in a flow of size ♯ops=50 with swa%=10, five operators may change their positions. In order to change the position of two operators (e.g., with swap), these two operators should be swapable. (We refer the interested reader to another paper for formal details on when swapping two operators is permitted [10].) So, the flow created in this example, should contain five operators that their schemata would allow a swap; e.g., one way to do this is to create operators that do not affect the schemata of nearby operators.

As another example, in the same flow and with rec%=5, we may add up to three recovery points, based on the following logic: a recovery point should be placed in a point where the cost of recovering from the closest existing recovery point (or from the beginning) is greater than the i/o cost for maintaining a new recovery point.

P[*fs*]: The probability of having function shipping for an operator *op* in the same engine or across all applicable engines is related to the number of available implementations *imp* for *op* in the same engine and across all applicable engines, respectively. For example, if there is a single implementation for an operator in an engine, then the probability for function shipping on the same engine is zero. If there are multiple implementations for an operator, then we can either consider (a) a uniform probability for function shipping or as typically happens in practice, (b) a weighted probability of using a specific implementation –either in the same engine or in different engines– based on the cost for using that implementation. The lower this cost, the higher the probability of choosing that implementation. For example, assuming that all available implementations imp_i, $i = 1, ..., n$ have a cost c_i, then the possible outcomes are as follows:

$$\circlearrowleft imp_i \ \rightarrow imp_j$$

That is, we either use the same implementation ($\circlearrowleft$) or we do function shipping and use a different implementation ($\rightarrow$). The probability of having function shipping: $P(FS(imp_i \rightarrow imp_j)) = 0$, when $i = j$. If $i \neq j$ and assuming that for k out of n possible implementations $c_i > c_l$, $l = 1..k$, then:

$$P(FS(imp_i \rightarrow imp_j)) = \frac{1}{\sum_{l=1..k} \frac{1}{c_l}} \times \frac{1}{c_j} \quad (1)$$

Thus, we may vary the collection of operators used in a flow and their implementations as well, in order to test different scenarios for function shipping.

P[*ds*]: Following a similar logic, the probability of data shipping depends on the configuration of data stores. When a flow has data stores in two different engines, the probability of data shipping is high. (If the flows involving these data stores converge, then the probability is one). On the other hand, if all data stores are placed in a single engine, the probability of data shipping is low. For the latter case, the probability for data shipping is not zero, because sometimes, we may decide to execute part of the flow on another engine even if we do not have a related data store there for performance reasons –e.g., when the host engine is much slower than a remote engine or it does not support an implementation needed.

Blocking/Pipeline Execution: An operator may work on a tuple-by-tuple basis allowing data pipelining or it may need the entire dataset, which blocks the data flow. This not only affects the flow execution, but also flow optimization. A flow optimizer could group together pipeline operators (even if the local costs would not improve with a possible swap) in order to boost pipeline parallelism. Thus, we need to tune the number of pipeline ♯op-p and blocking operators ♯op-b in a flow.

4.2 Engine Related Variants

As hybrid flows involve more than one engine, we take into account this angle too.

Operator Plurality: The average number ♯eg-imp of different implementations per operator in an engine.

Data Store Plurality: The average number ♯eg-dst of data stores related to a flow in an engine.

Engine Processing Type: The processing nature of an engine eg-typ, e.g., streaming, in-memory, disk-based processing.

Engine Storage Capability: The variant eg-str shows whether a processing engine uses a disk-based storage layer too –e.g., files in a local filesystem, HDFS, and so on– or whether all data resides in memory.

Engine Communication Capability: The communication methods supported in an engine eg-con, like specialized connectors to exchange data with another engine or simple import/export functionality.

Distributed Functionality: The variant eg-par shows if an engine is a parallel engine –like a parallel database or a Map-Reduce engine– or it is installed on a single node.

Node Plurality: The number of nodes ♯eg-nds where the engine is installed.
Threads: The average number of threads an engine may assign to an operation ♯eg-thd-op or to a flow fragment ♯eg-thd-fl.

CPU: The average number of CPU's ♯eg-cpu per node that the engine may use.

Memory: The average memory size ♯eg-mem per node that the engine may use.

Disk: The average disk size ♯eg-mem per node that the engine may use.

Disk Type: The type of the disk eg-disk that the engine may use; e.g., SSD's.

Failure Rate: The average number of failures ♯eg-flr that may happen in an engine node. This variant helps is simulating an environment for measuring flow fault tolerance.

4.3 Operator Related Variants

We consider tuning capabilities for flow operators.

Operator Type: The operator type op-tp. A typical number of operators involved in hybrid flows, as those described in the previous sections, is in the order of hundreds. It is very hard to agreed on a common framework without a classification of operators. In a previous approach to flow benchmarking, we proposed a taxonomy for ETL operators based on several dimensions, like the arity of their schemata (unary, n-ary, n-1, 1-n, n-m, etc.) and the nature of their processing (row-level, holistic, routers, groupers, etc.) [9]. Here, we consider the same taxonomy augmented by one dimension: physical properties, as captured by the variants below.

Parallelizable: The variant op-par captures whether an operator can be parallelized.

Code Flexibility: The average number of implementations ♯op-imp per operator.

Blocking/Pipeline: The variant op-bl captures the blocking or pipeline nature of an operator implementation.

In-Memory: The variant op-mem shows whether the operator functionality can be performed solely in memory.

CPU: The average number of CPU's ♯op-cpu per node that the operator may use.

Memory: The average memory size ♯op-mem per node that the operator may use.

Disk: The average disk size ♯op-mem per node that the operator may use.

Failure Rate: The average number of failures ♯op-flr that may happen during the operator execution.
All operator related variants V may be considered as flow related variants too, as an average number of V represented as ♯V. For example, a flow related variant is the average number of parallelizable operators in a flow ♯op-par. With op-tp, at the flow level we may determine the distribution of operators in a flow based on their types.

4.4 Data Related Variants

Finally, we need to tune the input data sets for hybrid flows.

Data Skew: The skew of data skew.

Data Size: The average input data size size.

Store Type: The variant st-tp indicates the storage type for a data set; e.g., flat file, stream, key-value store, RDF, database table. This variant may also be detailed by setting an average number per store type per flow, like ♯st-X%, where X takes any value from the domain: X = {fixed-width file, delimited file, HDFS file, relational table, XML file, RDF file, document, image, spreadsheet, stream}. For example, ♯st-file%=60 shows that 60% of the data stores involved in a flow will be delimited files (the default option for files). If there is no more information about store types, the remaining data stores are considered as database tables (this default value is tunable as fit).

Structure: The average structuredness of data as a percentage (str%); str%=0 shows unstructured data (like tweets) and str%=100 shows fully structured data (like tuples). Anything in between creates flows with mixed inputs; e.g., str%=x, where x<50, x% of ♯dst contain unstructured data and 100-x% of ♯dst contain structured data (if x>50, then the opposite percentage of ♯dst contain unstructured and structured data, respectively).

Data Per Engine: The average data size ♯eg-size stored per engine. We can also fine tune this at the node level: ♯eg-nd-size, the average data size residing per node of an engine.

Data Rate: The average rate in-rt that data arrive at the beginning of the flow.

5 Related Work

Optimization of Hybrid Flows. Previous work on hybrid flows has been done in two contexts: federated database systems and ETL engines. 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 [8], Multibase [2], and Pegasus [3]. ETL flows are hybrid in the sense that they extract from and load to database engines. Most ETL engines provide pushdown of some operators, e.g., filter, to the database engines [6] but the pushdown is a fixed policy and is not driven by cost-based optimization.

Optimizer Calibration. In the past, several researchers have used synthetic data and specially-crafted benchmark queries to calibrate query optimizers (e.g., as in [4,5,7]). This approach is especially attractive for federated database systems because the database engines can be treated as black boxes without exposing internal details. In general, this technique is limited to execution engines in

the same family, e.g., all relational stores or object stores. However, limited extensions to handle user-defined functions have been employed.

Benchmark Frameworks. For database systems, the suite of benchmarks devised by the Transaction Processing Performance Council are widely used in both industry and research. The benchmarks enable fair comparisons of systems due to the detailed specification of data and queries and the rules for conformance. Submitted results are audited for compliance. Because the benchmark is so well-understood, the associated datasets and queries are often used informally in research projects. The success of the database benchmarks inspired similar efforts in other domains. ETL benchmark efforts were begun [9,14], but to the best of our knowledge there has not been much progress. Several researchers have independently adapted TPC-H [13] or TPC-DS [12] for ETL benchmarks of their own design, but these are limited in scope and not designed for reuse.

An important requirement for benchmark frameworks is provenance and reproducibility. It must be possible to reproduce results and, to do this, a comprehensive accounting of the computing environment and input datasets used is needed. VisTrails [1] is a workflow management system for scientific computing that facilitates creation of workflows over scientific datasets and automatically tracks provenance. It is designed for change and tracks changes to workflows, including changes to operators and inputs. It also enable parameterized flow which makes it easy to scale a workflow to larger datasets. Such features are proving very useful to researchers and should be considered in the design of future benchmarks.

6 Conclusions

Enterprises are moving away from traditional back-end ETL flows that periodically integrate and transform operational data sources to populate a historical data warehouse. To remain competitive, enterprises are migrating to complex analytic data flows that provide near real-time views of data and processes and that integrate data from multiple execution engines and multiple persistent stores. We refer to these as multi-engine flows or hybrid flows. They are difficult to design and optimize due to the number of alternative, feasible designs; i.e., assignment of operators to execution engines. Our QoX optimizer is designed to optimize such hybrid flows. An important design factor is accurate estimation of data shipping and function shipping. This paper describes our approach to deriving cost formulae for the QoX optimizer. We have created a framework that utilizes microbenchmarks to collect metrics for data and function shipping, and we also list a set of interested variants. It is our hope that the emergence of hybrid flows may prompt reconsideration of work on industry standard benchmarks for analytic data flows. Our paper describes a step in this direction.

References

1. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T.: Managing the evolution of dataflows with VisTrails. In: ICDE Workshops, p. 71 (2006)
2. Dayal, U.: Processing queries over generalization hierarchies in a multidatabase system. In: VLDB, pp. 342–353 (1983)
3. Du, W., Krishnamurthy, R., Shan, M.C.: Query optimization in a heterogeneous DBMS. In: VLDB, pp. 277–291 (1992)
4. Ewen, S., Ortega-Binderberger, M., Markl, V.: A learning optimizer for a federated database management system. In: BTW, pp. 87–106 (2005)
5. Gardarin, G., Sha, F., Tang, Z.H.: Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In: VLDB, pp. 378–389 (1996)
6. Informatica: PowerCenter Pushdown Optimization Option Datasheet (2011)
7. Naacke, H., Tomasic, A., Valduriez, P.: Validating mediator cost models with disco. Networking and Information Systems Journal 2(5) (2000)
8. Roth, M.T., Arya, M., Haas, L.M., Carey, M.J., Cody, W.F., Fagin, R., Schwarz, P.M., Thomas II, J., Wimmers, E.L.: The Garlic project. In: SIGMOD, p. 557 (1996)
9. Simitsis, A., Vassiliadis, P., Dayal, U., Karagiannis, A., Tziovara, V.: Benchmarking ETL Workflows. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 199–220. Springer, Heidelberg (2009)
10. Simitsis, A., Vassiliadis, P., Sellis, T.K.: State-space optimization of ETL workflows. IEEE Trans. Knowl. Data Eng. 17(10), 1404–1419 (2005)
11. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: Optimizing analytic data flows for multiple execution engines. In: SIGMOD Conference, pp. 829–840 (2012)
12. TPC Council: TPC Benchmark DS (April 2012), `http://www.tpc.org/tpcds/`
13. TPC Council: TPC Benchmark H (April 2012), `http://www.tpc.org/tpch/`
14. Wyatt, L., Caufield, B., Pol, D.: Principles for an ETL Benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 183–198. Springer, Heidelberg (2009)