Optimization of Analytic Data Flows for Next Generation Business Intelligence Applications

Umeshwar Dayal, Kevin Wilkinson, Alkis Simitsis, Malu Castellanos, and Lupita Paz

HP Labs, Palo Alto, CA,USA
{umeshwar.dayal,kevin.wilkinson,alkis.simitsis, malu.castellanos,lupita.paz}@hp.com

Abstract. This paper addresses the challenge of optimizing analytic data flows for modern business intelligence (BI) applications. We first describe the changing nature of BI in today's enterprises as it has evolved from batch-based processes, in which the back-end extraction-transform-load (ETL) stage was separate from the front-end query and analytics stages, to near real-time data flows that fuse the back-end and front-end stages. We describe industry trends that force new BI architectures, e.g., mobile and cloud computing, semi-structured content, event and content streams as well as different execution engine architectures. For execution engines, the consequence of "one size does not fit all" is that BI queries and analytic applications now require complicated information flows as data is moved among data engines and queries span systems. In addition, new quality of service objectives are desired that incorporate measures beyond performance such as freshness (latency), reliability, accuracy, and so on. Existing approaches that optimize data flows simply for performance on a single system or a homogeneous cluster are insufficient. This paper describes our research to address the challenge of optimizing this new type of flow. We leverage concepts from earlier work in federated databases, but we face a much larger search space due to new objectives and a larger set of operators. We describe our initial optimizer that supports multiple objectives over a single processing engine. We then describe our research in optimizing flows for multiple engines and objectives and the challenges that remain.

Keywords: Business Intelligence, Data Flow Optimization, ETL.

1 Introduction

Traditionally, Business Intelligence (BI) systems have been designed to support off-line, strategic "back-office" decision making, where information requirements are satisfied by periodical reporting and historical queries. The typical BI architecture (Fig. 1) consists of a data warehouse that consolidates data from several operational databases and serves a variety of querying, reporting, and analytic tools. The back end of the architecture is a data integration pipeline for populating the data warehouse by periodically extracting data from distributed, often heterogeneous, sources such as online transaction processing (OLTP) systems; cleansing, integrating and transforming the data; and loading it into the data warehouse. The traditional data integration pipeline is a batch process, usually implemented by extract-transform-load (ETL) tools [26]. Designing and optimizing the ETL pipeline is still a challenging problem [e.g., 6, 21, 23]. After the data is cleansed and loaded into the data warehouse, it is then queried and analyzed by front-end reporting and data mining tools.

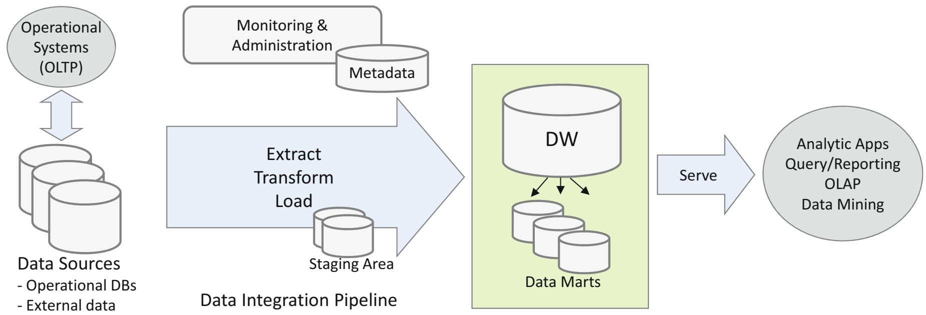

Fig. 1. Traditional business intelligence architecture

As enterprises become more automated, real-time, and data-driven, the industry is evolving toward Live BI systems that support on-line, “front-office” decision making integrated into the operational business processes of the enterprise. This imposes even more challenging requirements on the information pipeline. The data sources and data types are much more diverse: structured, unstructured, and semi-structured enterprise content, external data feeds, Web and Cloud-based data, sensor and other forms of streaming data. Faster decision making requires eliminating latency bottlenecks that exist in current BI architectures.

In this new architecture, as shown in Fig. 2, the back-end integration pipeline and the front-end query, reporting, and analytics operations are fused into a single analytics pipeline that can be optimized end-to-end for low latency or other objectives; for instance, analytics operations can be executed on “in-flight” streaming data before it is loaded into a data warehouse. Integration into the business processes of the enterprise requires fault tolerance, with little or no down-time. In general, optimizing the end-to-end pipeline for performance alone is insufficient. New quality objectives entail new tradeoffs; e.g., performance, cost, latency, fault-tolerance, recoverability, maintainability, and so on. We refer to these as *QoX objectives* [22]. Instead of a “one size fits all” engine, there may be many choices of engine to execute different parts of the pipeline: column and row store DBMSs, map-reduce engines, stream processing engines, analytics engines. Some operations are more efficiently executed in specific engines, and it may be best to move the data to the engine where the operation is most efficiently executed (*data shipping*). Other operations may have multiple implementations, optimized for different engines, and it may be better to leave data *in situ* and move the operation to the data (*function shipping*). The ETL flows, followed by querying, reporting, and analytics operations, are thus generalized to analytic data flows that may span multiple data sources, targets, and execution engines.

In this paper, we describe the problem of physical design and optimization of analytic data flows for the next generation BI applications. We model such flows as data flow graphs, whose nodes are data sources, targets, or operations on intermediate data. Given (a) a logical data flow graph, (b) optimization objectives for the flow, and (c) a physical infrastructure (data stores, processing engines, networks, compute nodes), the physical design problem is to create a graph that implements the logical flow on the physical infrastructure to achieve the optimization objectives.

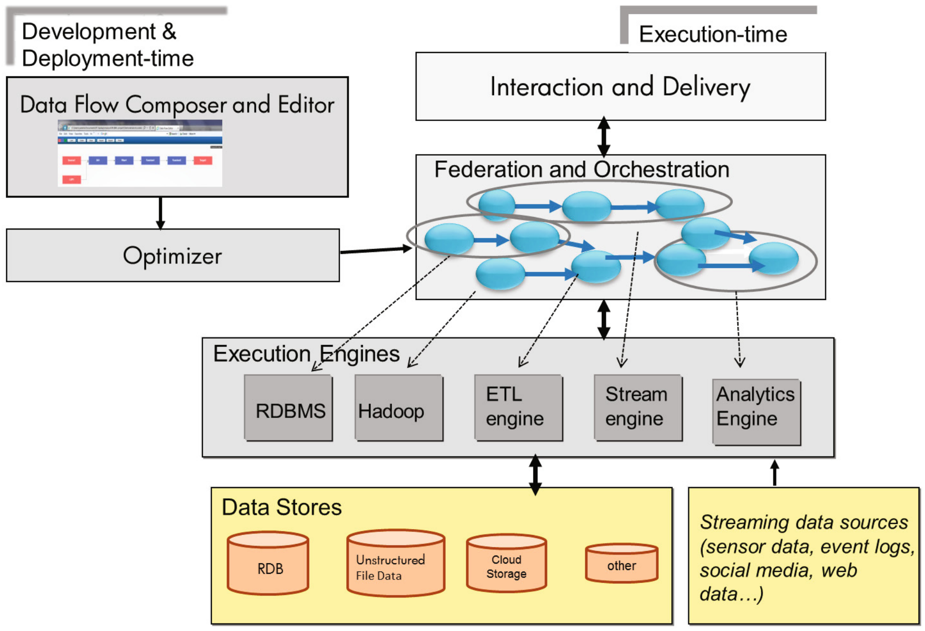

Fig. 2. Live BI system architecture

Example Scenario. We present a simple, example scenario to illustrate our approach. The scenario presumes a nationwide marketing campaign is conducted that promotes a small set of products. At the end of the campaign, a report is required that lists product sales for each product in the campaign. The input to the report is a modified version of the Lineitem fact table of TPC-H. This table lists, for each item in a purchase order, the quantity sold and the unit price. We also assume a dimension table that lists attributes for a marketing campaign. A synopsis of the database schema is shown below:

```
Lineitem:   orderKey, productKey, quantity, unitCost
Orders:     orderKey, orderDate
CmpmDim:    cmpnKey, productKey, dateBeg, dateEnd
RptSalesByProdCmpn:  productKey, cmpnKey, sales
```

The logical data flow to generate the report is shown in Fig. 3. We assume that the Lineitem table itself is created by periodically extracting recent line-item rows from OLTP databases in the various stores, taking their union, and then converting the production keys to surrogate keys. Similar extracts are needed for the Orders table and the dimension tables, but these are not shown. We acknowledge that a real-world flow would be much more complicated, e.g., multiple data sources and targets, extensive data cleaning, and so on. However, our purpose here is to illustrate our approach to optimization, so we have abstracted away many details in the flow.

To illustrate the new demands posed by Live BI, we augment our scenario by adding semi-structured content. We assume a Twitter feed is filtered for references to the enterprise's products and the resulting tweets are stored in the distributed file system of a Map-Reduce engine such as Hadoop. We perform a sentiment analysis on those Twitter feeds to obtain feedback on the public reaction to the campaign products and add that analysis to the campaign sales report.

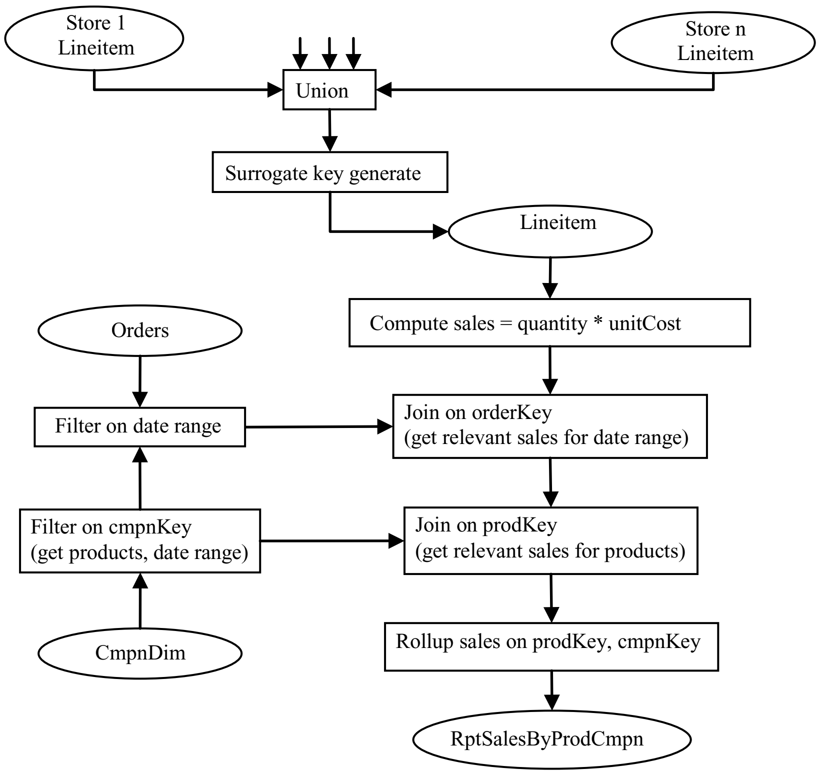
Store 1 Lineitem
Store n Lineitem
Union
Surrogate key generate
Lineitem
Compute sales = quantity * unitCost
Orders
Filter on date range
Join on orderKey (get relevant sales for date range)
Filter on cmpnKey (get products, date range)
Join on prodKey (get relevant sales for products)
CmpnDim
Rollup sales on prodKey, cmpnKey
RptSalesByProdCmpn

Fig. 3. Flow for RptSalesByProdCmpn

In this scenario, the Twitter feed includes any tweet that mentions the enterprise's products, and these tweets are passed to the sentiment analysis part of the flow. The logical data flow for this example is in Fig. 4. The sentiment analysis computation is shown here as a black box (later, in Section 2, we will expand this black box to show details of the sentiment analysis operations). The result of sentiment analysis is, for each tweet, a sentiment score for each *attribute* of a *topic*. For example, a tweet might have a weakly positive sentiment for the quality of a printer but a strongly negative sentiment for its price. To simplify, we assume a topic identifies a particular product, but in general a topic hierarchy or lattice could be used, e.g., a tweet might reference a particular printer model, a class of printers, or all printers sold by the enterprise. After sentiment analysis, the product identifiers in the tweets are replaced by their surrogate keys. The sentiment scores per tweet are then aggregated per product and time period so that they can be joined with the campaign sales report. In the rollup operation, the computation to aggregate the sentiment scores by attribute is assumed to be some user-defined function (defined by the marketing department) that assigns weight to the various sentiment values (e.g., the weight may depend on the influence score of the tweeter).

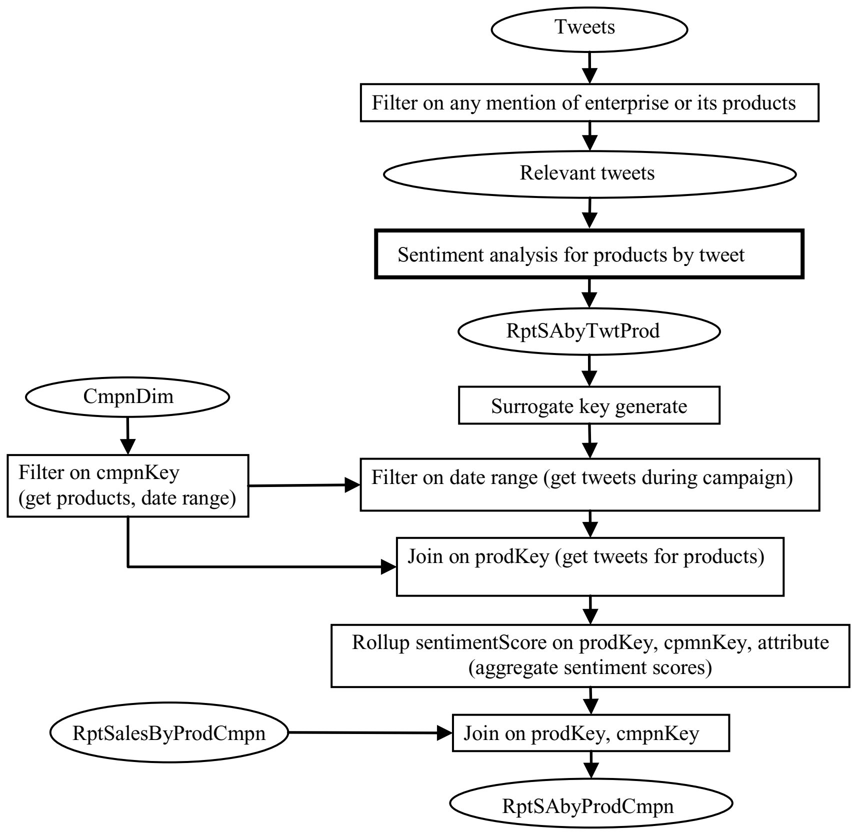
Tweets
Filter on any mention of enterprise or its products
Relevant tweets
Sentiment analysis for products by tweet
RptSAbyTwtProd
Surrogate key generate
CmpnDim
Filter on cmpnKey (get products, date range)
Filter on date range (get tweets during campaign)
Join on prodKey (get tweets for products)
Rollup sentimentScore on prodKey, cpmnKey, attribute (aggregate sentiment scores)
RptSalesByProdCmpn
Join on prodKey, cmpnKey
RptSAbyProdCmpn

Fig. 4. Expanded Flow for RptSAByProdCmpn

A synopsis of the additional tables used by this expanded flow is shown below.

Tweet:	`tweetKey, tweetUser, timestamp, tweetText`
RptSAbyTwtProd:	`tweetKey, timestamp, productKey, attribute, sentimentScore`
RptSAbyProdCmpn:	`productKey, cmpnKey, sales, attribute, sentimentScore`

In Section 2, we describe our framework for optimizing analytic data flows. Given a logical data flow graph of the kind shown in Fig. 3 and Fig. 4, and a set of QoX objectives, the optimizer produces a physical data flow that is optimized against those objectives. The optimizer assigns fragments of the flow graph to possibly different engines for execution. Since the choices made by the optimizer are cost driven, we have to characterize the execution of the fragments on different engines with respect to the different quality objectives. Our approach is to use micro-benchmarks to measure the performance of different engines on the various operators that occur in the analytic data flows. Section 3 describes some of our work on such micro-benchmarks. There are many remaining challenges in developing an optimizer for analytic data flows. After describing related work in Section 4, we list some of these challenges in Section 5.

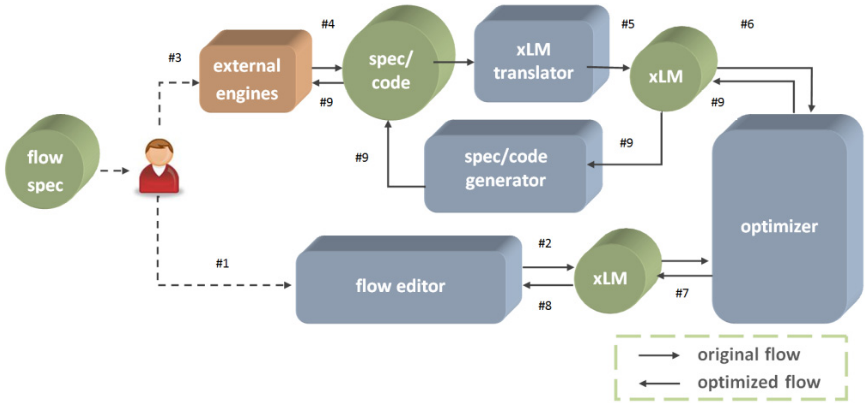

Fig. 5. QoX-driven optimization

2 Our Optimization Approach

Our optimization approach takes as input a flow graph, a system configuration, and some QoX objectives and produces as output a physical flow graph that is optimized according to the objectives (see Fig. 5).

The input logical data flow graph represents data sources, targets, and logical operations, together with annotations representing QoX objectives. Typical data sources are OLTP systems, flat files, semi-structured or unstructured repositories, sensor data, Web portals, and others. Typical targets are OLAP and data mining applications, decision support systems, data warehouse tables, reports, dashboards, and so on. The flows may contain a plethora of operations such as typical relational operators (e.g., filter, join, aggregation), data warehouse-related operations (e.g., surrogate key assignment, slowly changing dimensions), data and text analytics operations (e.g., sentence detection, part of speech tagging), data cleansing operations (e.g., customer de-duplication, resolving homonyms/synonyms), and others. How to design such a flow from SLAs and business level objectives is itself a challenging research and practical problem. Some work has been done in the past (e.g., [28]), but we do not elaborate further on this topic in this paper. Internally, each flow is represented as a directed acyclic graph that has an XML encoding in an internal language we call xLM.

A flow graph may be designed (edge #1 in Fig. 5) in a flow editor which directly produces xLM (edge #2) or in an external engine (edge #3); e.g., an ETL flow may be designed in a specific ETL engine and exported in some XML form (edge #4) (most modern ETL engines store flow metadata and execution information in proprietary XML files). An xLM parser translates these files (edge #5) and imports them into the QoX Optimizer (edge #6). The optimizer produces an optimized flow (edge #7) that is displayed back on the editor (edge #8), so that the user can see –and even modify– the optimized flow. In addition, an optimized file is translated into requests to the target execution engines (edges #9), which may include relational DBMSs, ETL engines, custom scripts, and map-reduce engines (such as Hadoop).

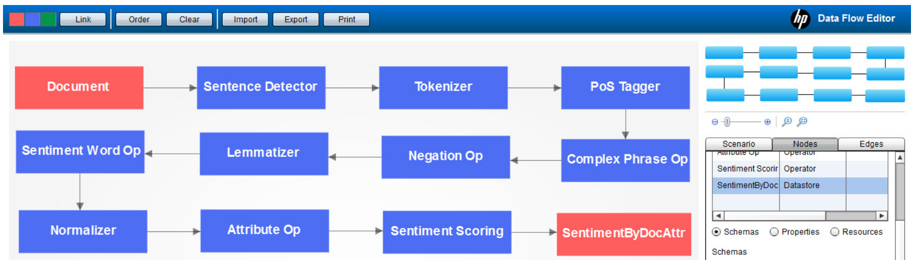

Fig. 6. Sentiment analysis flow graph

In past work, we have described a complete framework for optimizing flows for a variety of QoX objectives such as performance and fault tolerance [6]. Our approach to flow optimization involves both the logical and physical levels. For example, to optimize the logical flow in Fig. 3 for performance, the optimizer would compare the two joins over Lineitem (with the order keys and the product keys) and perform the most selective join first. It might also eliminate the sales computation task by combining it with the rollup task. At the physical level, if the Lineitem, Orders, and CmpnDim tables are all stored within the same RDBMS, the optimizer might perform the joins and rollup in the RDBMS rather than use an ETL engine.

To optimize the logical flow in Fig. 4 for fault tolerance, a recovery point might be inserted after the surrogate key generation, since sentiment analysis and surrogate key generation are time-consuming tasks that would have to be repeated in the event of a failure. As a more complicated example, suppose the combined flows of Fig. 3 and Fig. 4 were to be optimized for freshness. Then, it might be desirable to filter on the campaign dimension very early in the flow to significantly reduce the amount of useless data transferred. In this case, the lineitem extracts and the tweet stream could be filtered by campaign, product key, and date range. Note that the tweet dataset would then be much smaller. Hence, at the physical level, rather than storing it on the map-reduce engine, the optimizer may choose to extract it to a single compute node where the sentiment analysis could be performed faster, avoiding the higher overhead of the map-reduce engine.

Additional choices involve the granularity at which operations are considered by the optimizer. For example, sentiment analysis can be seen as a complex operator that in turn is composed by a flow of lower level operators corresponding to the different tasks in the analysis. Fig. 6 shows a flow graph for the sentiment analysis operation that was represented by a single node in Fig. 3, displayed on the canvas of the Flow Editor.

Treating this complex operation as a composition of other lower-level operators makes it possible for the optimizer to consider the possibility of executing the component operations on different engines. In our sentiment analysis flow, there is a Normalization operator (to reduce all variations of a word to a standard form) that does a look-up operation into a dictionary to retrieve the standard form of a given word. This kind of operation can be implemented in different ways: as a relational join, as a Unix script, as an ETL look-up operator, or as a program in Java (or some other programming language). Similarly, the Sentiment Word operator looks up a word in a lexicon to determine if the word is an opinion word. Assuming that different implementations exist for these look-up operators, the issue is to determine the best execution engine for each operator. Other operators in the sentiment analysis flow, for instance those which perform shallow natural language processing (NLP) tasks (sentence detection, tokenization, parts-of-speech tagging, lemmatization, and complex noun phrase extraction), are typically implemented as functions in NLP libraries.

2.1 QoX-Driven Optimizer

The QoX optimizer first works on a logical data flow graph and tries to optimize it for the specified QoX objectives. Then, the optimized flow can be further refined, and specific physical choices are made. There are several ways to optimize such a flow, even at the logical level. The most obvious objective is to meet an execution time window and produce accurate and correct results. Typically, optimizing a flow for objectives other than performance – e.g., fault tolerance, maintainability, or recoverability – might hurt performance. For example, for adding a recovery point we may have to pay an additional I/O cost. Thus, in general, we optimize first for performance and then consider strategies for other optimization objectives: adding recovery points, replication, and so on.

At the physical level, the optimized logical flow is enriched with additional design and execution details. For example, a logical operation (such as the join on prodKey) may have alternative physical implementations (nested loops, sort-merge, hash join). A specific algorithm may have different incarnations based on the execution engine; e.g., surrogate key assignments can have a sequential implementation on an ETL engine or a highly parallel implementation on a Hadoop engine. At this level, we need to specifically describe implementation details such as bindings of data sources and targets to data storage engines, and binding of operators to execution engines. Still, the physical plan should be independent of any specific execution engine. However, the optimization may use specific hooks that an engine provides in order to optimize better for that specific engine.

Internally, the QoX Optimizer formulates the optimization problem as a state space search problem. The states in this state space are flow graphs. We define an extensible set of transitions for producing new states. When a transition is applied to a state, a new, functionally equivalent, state is produced. Based on an objective function, we explore the state space in order to find an optimal or near optimal state. We have a set of algorithms to prune the state space and efficiently find an optimized flow satisfying a given set of objectives; i.e., an objective function.

An example objective function can be as follows:

$$OF(F, n, k,w): minimize\ c_{T(F)},\ where\ time(F(n, k)) < w$$

which translates to: minimize the execution cost, ‘c’, of a flow, ‘F’, such that its execution time window should be less than a specified size ‘w’ time units, its input dataset has a certain size, ‘n’, and a given number, ‘k’, of failures should be tolerated (see [23]).

The execution cost of a flow is a function of the execution costs of its operations. The choice of this function depends on flow structure. For example, the execution cost of a linear flow (i.e., a sequence of unary operations) can be calculated as the sum of the costs of its operations. Similarly, the execution cost of a flow consisting of a number of parallel branches is governed by the execution cost of the slowest branch. Cost functions for each operator capture resource cost, selectivity, processing rate, cost of data movement, and so on. Simple formulae for the execution cost of an operation can be determined based on the number of tuples processed by the operation; e.g., the execution cost of an aggregator may be $O(n \cdot \log n)$, where n is the size of the input dataset. More complex and accurate cost functions should involve output sizes (e.g., based on the operation’s selectivity), processing time (e.g., based on throughput), freshness, and so on. Deriving cost formulae for non-traditional operations (e.g., operations on unstructured data or user-defined analytic operations) that can appear in analytic data flows is a challenging problem. Section 3 describes an approach based on micro-benchmarks for obtaining cost formulae for individual operators. However, the optimization process is not tied to the choice of a cost model.

The set of state space transitions depends on the optimization strategies we want to support. For improving performance, an option is flow restructuring. With respect to the example of Fig. 3, we already mentioned pushing the most selective join early in the flow. Alternatively, one could consider partitioning the flow, grouping pipeline operations together, pushing successive operators into the same engine, moving data across engines or data stores, and so on. Typical transitions for improving performance are: swap (interchange the position of two unary operations), factorize/distribute (push a unary operation after/before an n-ary operation), and partition (add a router and a merger operations and partition the part of the flow between these two into a given number of branches). Example transitions for achieving fault tolerance are adding recovery points and replicating a part of the flow. Other transitions may be used as well so long as they ensure flow correctness. In [6], we described several heuristics for efficiently searching the state space defined by these transitions.

2.2 Extending the Optimizer to Multiple Engines

Our earlier work had focused on optimizing back-end integration flows, which we assumed were executed primarily on a single execution engine (typically, an ETL engine or a relational DBMS). We are now interested in optimizing analytic data flows that may execute on a combination of engines (ETL engines, relational DBMSs, custom code, Hadoop, etc.). This requires extending the physical level of the QoX Optimizer.

As an example, consider the flow depicted in Fig. 4. We can imagine three different execution engines being used to process this flow. The tweets are loaded into a map-reduce engine to leverage its parallel execution capabilities for the sentiment analysis task. The campaign dimension and campaign sales reports are stored in an RDBMS and must be retrieved for processing. Imagine the remaining flow (surrogate key, filter, joins, rollup) being processed by an ETL engine. The optimizer might choose to push some filtering tasks from the ETL engine down into the RDBMS (filter on campaign) and the map-reduce engine (filter on date range). As described earlierr, the sentiment analysis task is itself a series of sub-tasks and so the optimizer may choose to move some of its sub-tasks from map-reduce to the ETL engine.

In general, there are several engine options for executing an analytic data flow, and a particular task may have implementations on more than one execution engine. In order to automate the choice of an engine, first we need to characterize the execution of operations on different engines. For that, we perform an extensive set of micro-benchmarks for a large variety of operations.

Our use of micro-benchmarks is motivated by tools used to calibrate database query optimizers. Such tools measure the time and resource usage of various operators needed for query processing, such as comparing two character strings, adding two integers, copying a data buffer, performing random I/O, performing sequential I/O. The tools are run on each platform on which the database system will be deployed. The individual measurements are combined to estimate the cost of higher-level operations such as expression evaluation, table scans, searching a buffer, and so on. Section 3 provides an overview of our current work on micro-benchmarks.

Applying the micro-benchmark concept to our framework presents two challenges. First, our micro-benchmarks are high-level operations with parameters that create a large, multi-dimensional space (e.g., sort time might be affected by input cardinality, row width, sort key length, etc.). The benchmark cannot cover the entire parameter space, so point measurements must be taken. For an actual flow, it is likely that an operator's parameters will not exactly match a measured benchmark. So, interpolation is required, but it may reduce the accuracy of the estimate.

A second challenge is that the ultimate goal is to estimate the cost of a data flow, not the cost of individual operators. The operators are coupled through the data flow and may interact in complex ways. A method is needed to compose the micro-benchmarks for individual operators to estimate the cost of a flow. We previously mentioned how the cost of a parallel flow is determined by the slowest branch. As another example, consider a flow of two operators, one producer and one consumer. If producer and consumer process data at similar rates and do not share resource, then the cost of the flow is a simple linear combination. However, if they run at different rates, the slower operator meters the flow. Additionally, if they share resources, the interaction must be considered when composing the individual micro-benchmark results. Addressing these challenges is a current area of research.

3 Micro-benchmarks for Performance

As described in Section 2, the goal of the optimizer at the physical level is to decide on the appropriate execution engine to process fragments of a flow graph in order to achieve the QoX objectives for the entire flow. The optimizer considers both the storage location of data (and the associated execution engine, e.g., RDBMS for tables, map-reduce for DFS, etc.) as well as the execution engines available for processing a flow graph (e.g., ETL engines, custom scripts, etc.). The optimizer may choose to perform data shipping, in which data is moved from the storage system of one execution engine to another, or it may choose to perform function shipping, in which a task is pushed down to be performed in the execution engine where the data is stored. A task that has multiple implementations is said to be polymorphic, and the optimizer must evaluate each implementation relative to the flow objectives.

In this section, we describe how the optimizer characterizes the implementation of a task relative to an objective. The approach is to use micro-benchmarks to measure the performance of a task at various points in the parameter space. The focus here is just on performance but the same approach can be used for other objectives. The performance curves generated from the micro-benchmarks can then be compared to choose the best implementation of a task for a specific flow graph configuration. In the first sub-section, we describe how micro-benchmarks are applied for conventional (ETL, relational) operators. In the second sub-section, we describe how the same approach can be extended for the new types of operators of Live BI. In particular, we show how micro-benchmarks can be used for the text analytics operators that comprise the sentiment analysis flow.

3.1 Micro-benchmarks for Conventional Operators

We study how several parameters of the flow and the system configuration affect the design choice. Example flow parameters to consider are: data size, number and nature of operations (e.g., blocking vs. non-blocking), flow and operation selectivity, QoX objectives such as degree of replication vs. desired fault-tolerance and freshness, input data size and location (e.g., if a mapping table is in a file or in the database). Example system configuration parameters include: network load and bandwidth, cluster size and resources, cluster node workload, degree of parallelism supported by the engine, and so on.

Knowing how each operation behaves on different engines and under various conditions, we may determine how a combination of operations behaves, and thus, we may decide on how to execute a flow or different segments of the flow. To illustrate our approach, we discuss example alternative designs for a sequence of blocking operations typified by the sort operation. Due to their blocking nature, as we increase the number of such operations, on a single-node, the flow performance linearly decreases. We experimented with different parallel implementations for improving the performance of the flow. Candidate choices we compared are: Unix shell scripts (os-sort), an ETL engine (etl-sort), a parallel dbms (pdb-sort), and Hadoop (hd-sort). Due to space considerations, we discuss the os-sort and hd-sort methods and present example tradeoffs among os-sort, hd-sort, and pdb-sort.

We implemented os-sort as a combination of C code and shell scripts in Unix, running directly on the operating system. First, the data file is split in equal-sized chunks based on the formula: *file size / #nodes* (any possible leftover is added to the first chunk). Then, each chunk is transferred to a different remote node, where it is sorted. Assuming we have a series of n blocking operators, each sort operator i (where i = 1…n) sorts the i-th data field; if the number of fields is less than n, then the (n + 1)-st operator goes back to the first field. In doing so, we eliminate the impact of cache memory in our experiments. Next, the sorted data chunks are transferred back to a central node and merged back in a hierarchical manner. Intentionally, we tried to avoid pipelining as much as we could, in order to make a fair comparison with both the etl-sort and hd-sort, where there is no pipelining between blocking operators. In practice, therefore, os-sort could perform better than our results show; however, the trends shown in our findings do not change. hd-sort is executed as Hadoop code. We tested different variations namely in-house developed user-defined functions (udf), Pig scripts, and JAQL scripts. Although we did observe differences in terms of absolute numbers, the behavior of this approach compared against the other strategies is not affected much. Comparing different Hadoop implementations is not amongst our goals; so here, we just present the generic trend (based on average numbers) and explain the functionality using only the Pig language. As an example of a series of blocking operators, we may write the following script in Pig:

```
sf$sf = load 'lineitem.tbl.sf$sf'
      using PigStorage('|') as (f1,f2,...,f17);
ord1_$nd = order sf$sf by f1 parallel $nd;
...
ord10_$nd = order ord9_$nd by f10 parallel $nd;
store ord$op_$nd into 'res_sf$sf_Ord$op-$nd.dat'
      using PigStorage('|');
```

Table 1. Statistics for TPC-H lineitem

SF	1	10	100
Size (GBs)	0.76	7.3	75
Rows (x10^6)	6.1	59.9	600

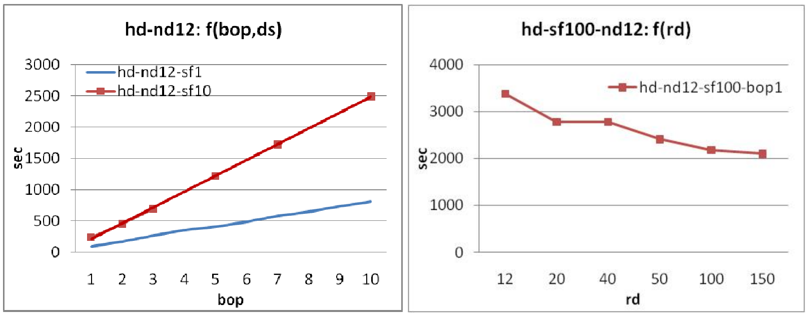

Fig. 7. Execution of blocking operation in Hadoop

The parameters used in this script, $sf, $nd, $op, stand for the scale factor of the lineitem datafile, the number of nodes, and the number of operators (sorters in this example), respectively.

For the experiments, we used synthetic data produced using the TPC-H generator. We experimented with varying data sizes using the scale factors 1, 10, and 100. Example statistics for the lineitem table are shown in Table 1.

Fig. 7 shows how Hadoop implementations behave for blocking operations (hd-sort). The left graph shows that as the number of blocking operations increases, flow performance is negatively affected. In fact, performance becomes worst if at the same time the data size increases too. Thus, for large files, having a series of blocking operations executed one after the other becomes quite costly. One optimization heuristic we can use for improving Hadoop performance is to increase parallelism by increasing the number of reduce tasks. The right graph shows how performance is improved when we increase the number of reducers (up to a certain point) whilst executing a single blocking operation for a SF=100 datafile (~75GBs).

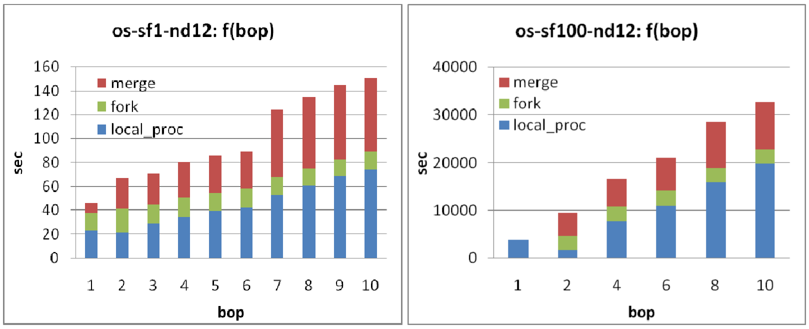

Fig. 8. Execution of blocking operation with shell scripts

Fig. 8 shows performance measures for flows executed as shell scripts in Unix (os-sort). The left and right graphs show an analysis of performance measures for two datafiles sized 0.76 GB and 75 GBs or SF=1 and SF=100, respectively. The blue part of each bar (bottom part) represents the time spent on each remote node, while the green (middle) and red (upper) part of each bar represent the time spent for distributing and merging back, respectively, data on the master node. As we increase the number of blocking operations each remote node has more processing to do. On the master node, distributing data is not affected by the number of remote nodes (each time we need to create the same number of chunks), while the merge time increases as we increase the number of blocking operations. This happens because each blocking operator essentially sorts data on a different field: the first on the first field and the N-th on the N-th field. Thus, each time we have to merge on a field that is placed deeper in the file, and thus merge has to process more data before it reaches that field.

Fig. 9 compares os-sort, hd-sort, and pdb-sort for executing a series of blocking operations (1 to 10). pdb-sort ran on a commercial parallel database engine. Starting from the top-left graph and going clockwise, the graphs show results of the three methods on small-sized (SF=1), medium-sized (SF=10), and large-sized (SF=100) data files. In all cases, pdb-sort is faster than os-sort and hd-sort. Hence, if our data resides inside the database, there is no reason to sort data outside. However, if our data is placed outside the database (e.g., data coming from flows running elsewhere like the results of hadoop operations on unstructured data) we have to take into account the cost of loading this data into our parallel database. This cost increases with data size and for this case, the total time (load+sort) is shown in the graphs as the green line (pdbL). In this scenario, there are some interesting tradeoffs. For large datasets and for a small number of sort operations, it might make sense to use hadoop. For medium sized datasets, it might worthwhile to pay the cost of loading the data into a database. For small datasets, if the data is not in the database, it is too expensive to run pdb-sort; then, it is better to sort outside either using os-sort (e.g., an ETL tool, custom scripts) or even hadoop (up to a certain number of sort operations).

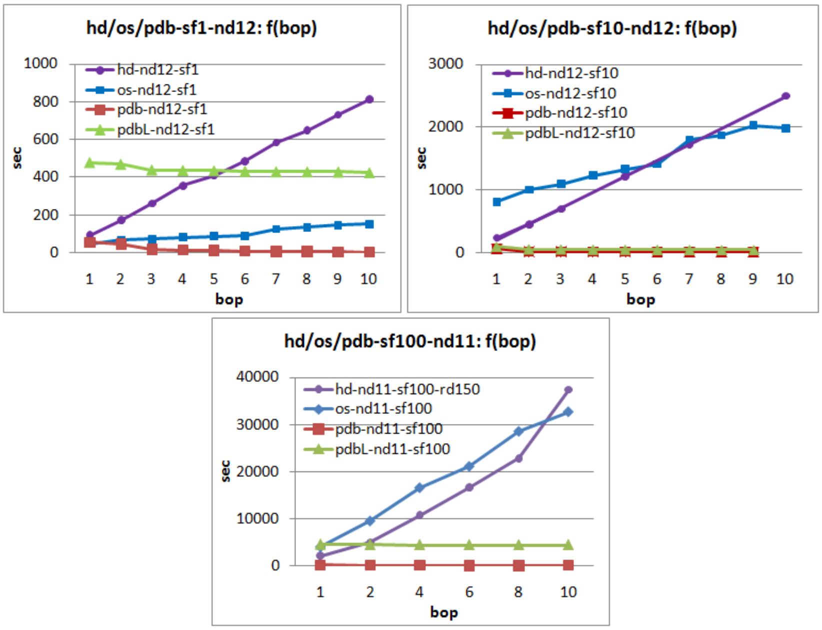

Fig. 9. Comparison of flow execution using Hadoop and shell scripts

Between os-sort and hd-sort, it seems that, for large data files and up to a certain number of blocking operations, it is better to use the hd implementation. After that number, the Hadoop reduce tasks become quite expensive and hd is not the best option any more. On the other hand, the trend changes for smaller data files. For small files, hd-sort is always the worst case, where for medium-sized files there is a crossover point. If we have high freshness requirements, then typically we have to process smaller batches. In such cases, Hadoop may become quite expensive. If we have to process larger batches (e.g., when we have low freshness requirements or the source data is updated less frequently), then Hadoop might be a very appealing solution.

In the same way, we may perform similar micro-benchmarks for other operations. We also need to cover the other parameters mentioned earlier and we need to define and perform micro-benchmarks for other QoX objectives in addition to performance. Using the micro-benchmarks for optimization poses further challenges such as *interpolation* (e.g., micro-benchmark measures 1MB sort and 10MB sort, but the actual flow has 3MB sort) and *composition* (estimating the performance of segments of a flow given the performance of individual operations in the flow).

3.2 Micro-benchmarks for Unconventional Operators

For optimizing data analytic flows for performance and other objectives, we need to consider operations that significantly differ from traditional relational and ETL operations, such as operations used in text analytics flows. This section discusses our approach for benchmarking the operators that occur in the sentiment analysis flow of Fig. 4. As discussed earlier, these operators are complex and can be benchmarked at two levels of granularity: as a single black box operator (as shown in 4) or as a flow of individual operators (as shown in 6).

To estimate cost functions for different implementations of each operator, we execute them on a set of unstructured documents of different sizes to obtain a series of point measurements and then apply regression to these points to learn an interpolation function.

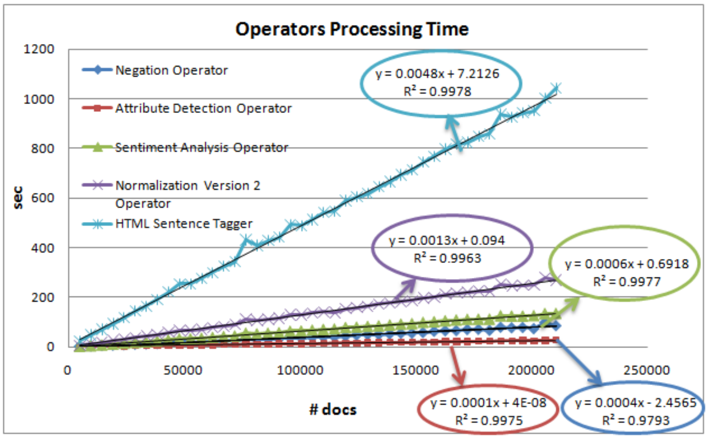

Fig. 10. Linear regression of execution costs

The initial experiments were on a single node and used dataset sizes from a few thousand documents up to 100,000. In all these runs, the operators of the sentiment analysis flow of our sample scenario exhibited a linear behavior. Thus, we used linear regression to approximate the data points to a line with minimum error, as the R-squared values in Fig. 10 show. These functions can be plugged into the QoX Optimizer so that it can interpolate to other values.

Further experiments with larger datasets revealed that an important parameter that affects this linear behavior is memory size. The individual operators have different memory requirements and consequently different sensitivity to the memory size. There is an inflection point at which the linear behavior changes to exponential, due to increased paging. The inflection point at which this change in behavior occurs for a given operator depends on the amount of available memory. In particular, for the sentiment analysis operators implemented in Java, the JVM heap size determines the location of the inflection point, which varies from operator to operator as depicted in the left three charts of Fig. 11 for two example operations, tokenizer and attribute detection, and for the entire sentiment analysis flow. (The bottom-left chart shows the behavior when the entire sentiment analysis is implemented as a single, black-box operator in a single JVM.) However, for the same experiments ran on a 12-node Hadoop cluster, the behavior remained linear past the single-node inflection point range of 150K to 190K documents for the different operators. We went up to 30 million documents and the behavior was still linear as shown in the two middle charts of Fig 12. Consequently, the overall processing time for the entire flow also remained linear.

Another set of experiments focused on a performance comparison among different implementations: (i) a single node implementation; (ii) a distributed implementation on Hadoop, in which the entire sentiment analysis flow was implemented as a single map task that Hadoop could distribute among the nodes of the cluster; and (iii) a distributed implementation without Hadoop.

Fig. 12 shows the results for different dataset sizes and the three different implementations. As for the conventional operators, we observe that for small datasets it is too costly to use Hadoop due to its startup cost. The distributed implementation without using Hadoop outperforms the Hadoop implementation because we partitioned the data set uniformly on all 12 nodes in the cluster. The Hadoop implementation, on the other hand, used the default block size of 64MB, and hence used at most 3 nodes. Fig. 13 illustrates this point. In the left chart of Fig. 13, we can distinguish two regions: one where the execution time of the operator grows linearly in the dataset size and the other, where the execution time stabilizes. For small datasets, a single partition is enough and consequently only one map task in one node is needed. The execution time of the task is proportional to the size of the partition. However, as the datasets get larger, more map tasks are created to process in parallel the various partitions, and the execution time stabilizes, not depending any more on the dataset size. This suggests that if it selects the Hadoop implementation, the optimizer will need to control the block size depending on the dataset size. The right graph shows that within the stable region, as the dataset size gets larger, more map tasks are needed to process the larger number of partitions.

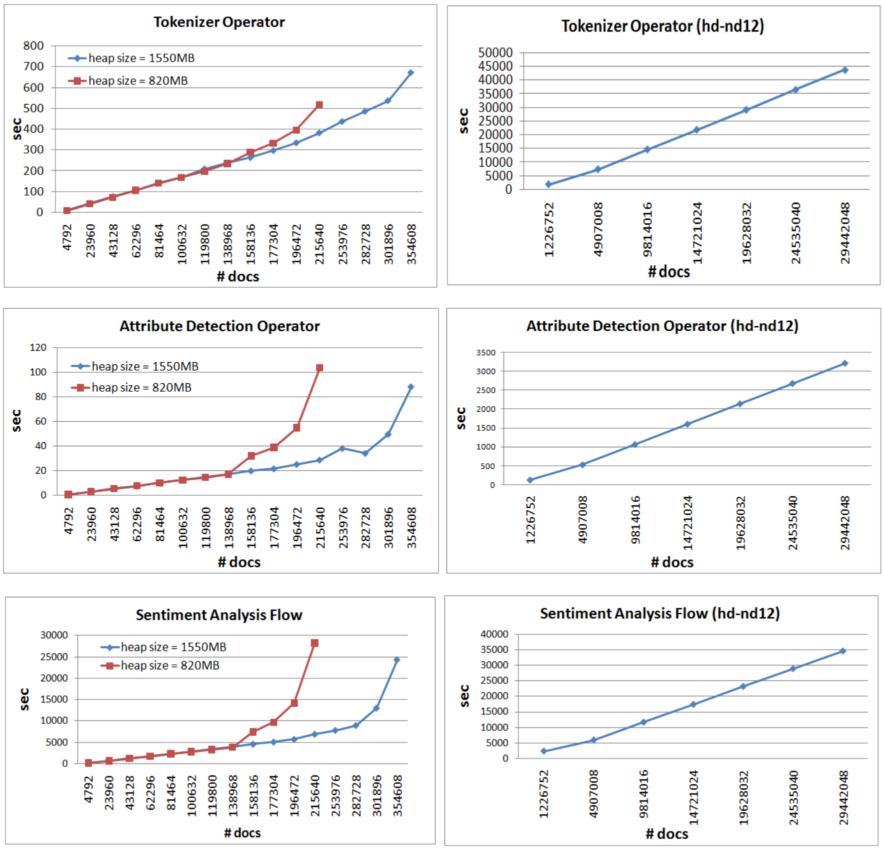

Fig. 11. Execution cost inflection point sensitiveness to JVM heap size

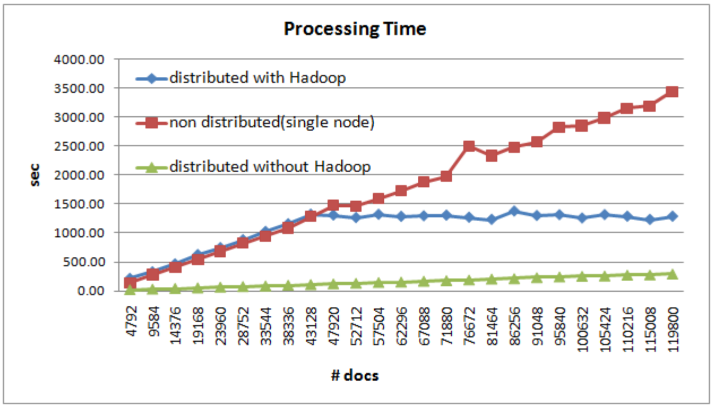

Fig. 12. Execution times for sentiment analysis operator

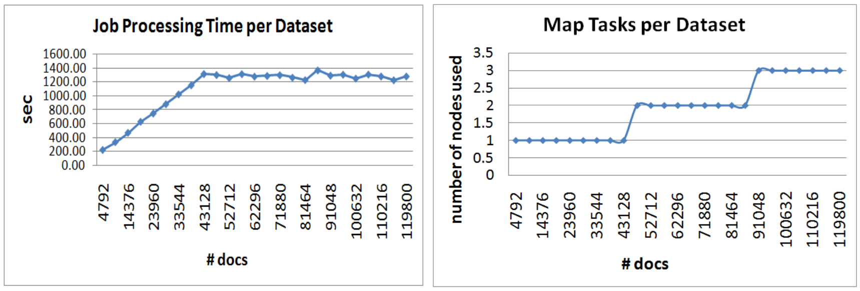

Fig. 13. Performance of sentiment analysis operator implemented in Hadoop

4 Related Work

To the best of our knowledge, there is no prior work on optimizing end-to-end analytic data flows, and very little work even on optimizing back-end integration flows. Off-the-shelf ETL engines do not support optimization of entire flows for multiple objectives (beyond performance). Some ETL engines provide limited optimization techniques such as pushdown of relational operators [10]) but it is not clear if and how these are tied to optimization objectives. Beyond our own QoX approach to integration flow optimization, research on ETL processes and workflows has not provided optimization results for multiple objectives. An optimization framework for business processes focuses on one objective and uses a limited set of optimization techniques [27]. Query optimization focuses on performance and considers a subset of operations typically encountered in our case [e.g. 9, 13, 18, 19]. Also, we want the optimizer to be independent of the execution engine; in fact, we want to allow the optimized flow to execute on more than one engine. Research on federated database systems has considered query optimization for multiple execution engines, but this work, too, was limited to traditional query operators and to performance as the only objective; for example, see query optimization in Garlic [8,16], Pegasus [7], and Multibase [5].

Several research projects have focused on providing high-level languages that can be translated to execute on a map-reduce engine (e.g., JAQL [3], Pig [15], Hive [24]). These languages offer opportunities for optimization. Other research efforts have generalized the map-reduce execution engine to create a more flexible framework that can process a large class of parallel-distributed data flows (e.g. Nephele [2], Dryad [11], CIEL [14]). 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 [4], Skywriting in CEIL). None of these projects addresses data flows that span different execution engines. HadoopDB is one example of a hybrid system in which queries span execution engines [1]. 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.

Several benchmarks do exist, but so far, none is specifically tailored for generalized analytic data flows. TPC has presented a successful series of benchmarks. TPC-DS provides source and target schemas, but the intermediate integration process is quite simplistic (it contains a set of insert and delete statements mostly based on relational operators) [25]. TPC-ETL is a new benchmark that TPC is currently working on and it seems to focus on performance, but no further information has been released yet. Another effort proposes an ETL benchmark focusing on modeling and performance issues [20]. Several benchmarking and experimental efforts on map-reduce engines have been presented, but so far, these focus mainly on performance issues [e.g., 12, 17].

5 Conclusions and Future Work

In this paper, we have addressed the challenges in optimizing analytic data flows that arise in modern business intelligence applications. We observe that enterprises are now incorporating into their analytics workflows more than just the traditional structured data in the enterprise warehouse. They need event streams, time-series analytics, log file analysis, text analytics, and so on. These different types of datasets are best processed using specialized data engines; i.e., *one size does not fit all.* Consequently, analytic data flows will span execution engines. We sketched our previous work on QoX-driven optimization for back-end information integration flows, where the quality objectives include not just performance, but also freshness, fault-tolerance, reliability, and others. This paper outlines how this approach can be extended to optimizing end-to-end analytic data flows over multiple execution engines. We described the results of initial micro-benchmarks for characterizing the performance of both conventional (ETL and database) operations and unconventional (e.g., text analytic) operations, when they are executed on different engines.

Many challenges remain, and we hope to address these in future work. These include:

- defining and implementing micro-benchmarks for additional representative operations, including event and stream processing, front-end data mining and analytic operations;
- developing interpolation models and interaction models for estimating the cost of complete flows;
- optimization strategies at the physical level for assigning segments of the flow to execution engines; and
- extending the benchmarks, cost models, objective functions, and optimization strategies to QoX objectives other than performance.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: Hadoop DB: An Architectural Hybrid of Map Reduce and DBMS Technologies for Analytical Workloads. PVLDB 2(1), 922–933 (2009)

2. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/PACTs: a Programming Model and Execution Framework for Web-Scale Analytical Processing. In: SoCC, pp. 119–130 (2010)
3. Beyer, K., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C.C., Ozcan, F., Shekita, E.: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In: VLDB (2011)
4. Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. PVLDB 1(2), 1265–1276 (2008)
5. Dayal, U.: Processing Queries over Generalization Hierarchies in a Multidatabase System. In: VLDB, pp. 342–353 (1983)
6. Dayal, U., Castellanos, M., Simitsis, A., Wilkinson, K.: Data Integration Flows for Business Intelligence. In: EDBT, pp. 1–11 (2009)
7. Du, W., Krishnamurthy, R., Shan, M.-C.: Query optimization in heterogeneous DBMS. In: VLDB, pp. 277–291 (1992)
8. Haas, L., Kossman, D., Wimmers, E.L., Yang, J.: Optimizing Queries across Diverse Data Sources. In: VLDB, pp. 276–285 (1997)
9. Han, W.-S., Kwak, W., Lee, J., Lohman, G.M., Markl, V.: Parallelizing query optimization. PVLDB 1(1), 188–200 (2008)
10. Informatica. PowerCenter Pushdown Optimization Option Datasheet (2011), `http://www.informatica.com/INFA_Resources/ds_pushdown_optimization_6675.pdf`
11. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In: EuroSys (2007)
12. Jiang, D., Chin Ooi, B., Shi, L., Wu, S.: The Performance of MapReduce: An In-depth Study. PVLDB 3(1), 472–483 (2010)
13. Lohman, G.M., Mohan, C., Haas, L.M., Daniels, D., Lindsay, B.G., Selinger, P.G., Wilms, P.F.: Query Processing in R*. In: Query Processing in Database Systems, pp. 31–47 (1985)
14. Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., Hand, S.: CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In: USENIX NSDI (2011)
15. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a Not-so-foreign Language for Data Processing. In: SIGMOD, pp. 1099–1110 (2008)
16. 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)
17. Schad, J., Dittrich, J., Quiané-Ruiz, J.-A.: Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. PVLDB 3(1), 460–471 (2010)
18. Sellis, T.K.: Global Query Optimization. In: SIGMOD, pp. 191–205 (1986)
19. Sellis, T.K.: Multiple-Query Optimization. TODS 13(1), 23–52 (1988)
20. 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)
21. Simitsis, A., Vassiliadis, P., Sellis, T.K.: Optimizing ETL Processes in Data Warehouses. In: ICDE, pp. 564–575 (2005)
22. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: QoX-driven ETL design: Reducing the Cost of ETL Consulting Engagements. In: SIGMOD, pp. 953–960 (2009)
23. Simitsis, A., Wilkinson, K., Dayal, U., Castellanos, M.: Optimizing ETL Workflows for Fault-Tolerance. In: ICDE, pp. 385–396 (2010)

24. Thusoo, A., Sen Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive - a Petabyte Scale Data Warehouse Using Hadoop. In: ICDE, pp. 996–1005 (2010)
25. TPC. TPC-DS specification (2011), `http://www.tpc.org/tpcds/spec/tpcds1.0.0.d.pdf`
26. Vassiliadis, P., Simitsis, A.: Extraction, Transformation, and Loading. In: Encyclopedia of Database Systems, pp. 1095–1101 (2009)
27. Vrhovnik, M., Schwarz, H., Suhre, O., Mitschang, B., Markl, V., Maier, A., Kraft, T.: An Approach to Optimize Data Processing in Business Processes. In: VLDB, pp. 615–626 (2007)
28. Wilkinson, K., Simitsis, A., Castellanos, M., Dayal, U.: Leveraging Business Process Models for ETL Design. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 15–30. Springer, Heidelberg (2010)