Graceful Logic Evolution in Web Data Processing Workflows

Christopher Olston
Yahoo! Research

ABSTRACT

Data processing workflows evolve over time. For example, operators of web information extraction workflows change them continually by retraining classifiers, adjusting confidence thresholds, extracting additional structured data fields, and incorporating new information sources. In most workflow systems, even minor workflow logic changes trigger recomputation of derived data products from scratch. When data volumes are large this approach is grossly wasteful and generates huge latency hiccups.

This paper develops a model and implementation of a workflow system that processes data incrementally in the face of workflow logic evolution, in addition to data evolution. Experiments on real data demonstrate large performance gains from this kind of incremental processing.

1. INTRODUCTION

Data-intensive workflows are important in many domains, including e-science and web data processing. When possible, workflows incorporate new and updated input data incrementally, as in view maintenance and data stream processing, to update the output data without re-processing all input data from scratch. Hence *data evolution* is handled gracefully, without introducing major hiccups in resource usage or output data freshness.

In addition to data evolution, it is common for the logic of the workflow to change over time. For example, ad-hoc scientific workflows are subject to a series of adjustments (some of them minor, some major) as the data analysis process is refined [11]. This paper was inspired by web information extraction [7] workflows, which continually evolve by retraining classifiers, adjusting confidence thresholds, extracting additional structured data fields, and incorporating new information sources.

To our knowledge no existing systems handle workflow *logic evolution* gracefully. For data processing purposes, a modified workflow is treated as a new workflow, and data is re-processed from scratch. Derived data sets created by the old version of the workflow are either re-used as-is, or are thrown out—no matter how closely related to the output of the modified workflow. Meanwhile, new data accumulating on the inputs is forced to wait, resulting in an unacceptable freshness bubble. The waste and delay incurred by workflow modifications becomes arbitrarily bad over time, as the amount of already-processed data grows.

1.1 Problem Statement

This paper focuses on large-scale data processing workflows, and in particular ones that:

- follow the "synchronous data-flow" (SDF) model of computation [19],
- are deployed for an extended period of time in a production environment, and
- incorporate newly-arriving input data in large batches using incremental algorithms[1].

Examples of this scenario include continuously-running SDF scientific workflows [19], incremental extract-transform-load (ETL) processes [16], web information extraction workflows [7], and continuous bulk processing models for map-reduce-like environments [2, 15, 18, 21].

In that context, the problem addressed by this paper is the following: Consider workflow W that transforms input data I into output data O, and along the way stores intermediate data L—i.e. $W(I) = \langle L, O \rangle$. The standard incremental processing goal is that if I changes to $I' = f(I, \Delta I)$ for some data update ΔI and update application function $f(\cdot)$, such that $W(I') = \langle L', O' \rangle$, we wish to produce O' (and perhaps also L', in anticipation of future incremental processing) at minimal cost from some combination of I, ΔI, I', L, and O. Our goal is to handle the $I \rightarrow I'$ case, and additionally handle the case in which W is changed to a new (but related) workflow W' where $W'(I) = \langle L'', O'' \rangle$, by finding a minimal-cost way to produce O'' (and L'', perhaps) from I, L and O. Moreover, we wish to handle simultaneous evolution of data and logic, i.e. compute $W'(I')$ incrementally.

[1]The incremental algorithms can be *stateless* (e.g. extract noun phrases from every newly-crawled web page) or *stateful* (e.g. maintain a running total of the number of occurrences of each extracted phrase).

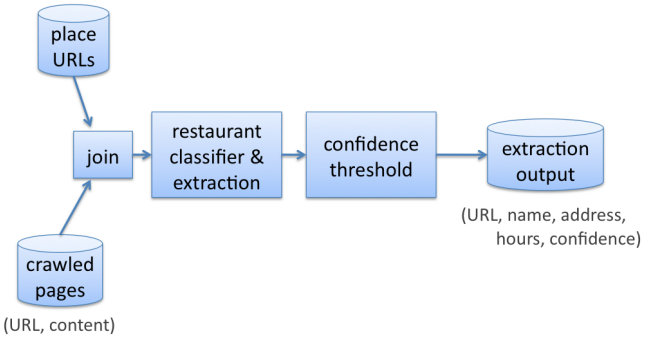

Figure 1: Restaurant extraction workflow fragment.

1.2 Related Work

Prior work on evolving business and scientific processes [4, 6, 10, 12] focused on avoiding re-running steps that have already been run by a prior incarnation of the process, i.e. re-using derived data sets. Under that approach, data sets that do not exactly match what the revised workflow would produce are discarded. This paper considers how to carry old derived data sets forward by updating them to bring them into alignment with the revised process description (workflow). This capability is important when derived data sets grow very large, and the updates required are cheap relative to full re-computation. The presence of large derived data sets that are cheaper to update than recompute motivated the *incremental view maintenance* problem [13], in which derived data sets are updated in response to changes to input data sets (not workflow logic changes).

Many aspects of our work, e.g. temporal data management and schema evolution, have been studied extensively in prior work [9, 22], but not to our knowledge in the context of incremental updating of data produced by evolving workflows. Our aim is not to innovate on these topics *per se*, but rather to use simple temporal data and schema evolution models as part of a novel framework for evolving data-intensive workflows.

In the context of schema evolution, there is work on incremental data updating for workflows that perform schema restructuring, e.g. [17]. However, these are static workflows whose purpose is to transform data from one schema to another (e.g. for data integration), not workflows that evolve over time.

1.3 Example

Consider the workflow fragment shown in Figure 1, which is based on a real web information extraction workflow from the Yahoo! Purple SOX project [5]. The workflow fragment takes a set of URLs that a prior classifier (not shown) has determined may correspond to places (e.g. they main contain an address and/or a map), and reads the corresponding page content from a web crawl repository. The content is fed to a restaurant classifier and extractor, which determines whether a page describes a restaurant, and if so extracts features such as name, address, and hours of operation; each output restaurant record is assigned a confidence score. Lastly, restaurant records are filtered according to a confidence threshold (e.g. only records with confidence at least 0.9 are retained).

This workflow fragment may undergo the following changes over time:

- Raise or lower the confidence threshold, to adjust the incidence of false positives/negatives in the output.
- Change the way one of the extracted fields (e.g. hours of operation) is computed, while leaving the other fields unaltered.
- Add or remove fields, e.g. restaurant menus.

In principle, many workflow modifications can be handled by transforming the existing output data. We call such a transformation a *migration plan*. For example, an efficient migration plan for raising the confidence threshold simply filters the output using the new threshold.

The case of lowering the threshold is harder: one possible migration plan is to re-process the input data after anti-joining it with URLs in the old output, and adding the new output records to the old ones. (If the original threshold was 0.9 and the new threshold is 0.8, this plan avoids re-processing items with confidence above 0.9; if most of the cost is in the extraction step (vs. classification), the savings may be large.)

Upgrading the hours-of-operation extraction module can be handled by re-processing the entire input but only running that extractor, and emitting a replacement column for the output data. Addition of a menu field can be handled similarly. A migration plan for removing a field is analogous to raising the confidence threshold, except that it removes a column from the output rather than removing rows.

1.4 A Framework for Evolving Workflows

It should be evident from the example in Section 1.3 that some migration plans could easily be generated automatically, e.g. exploiting easy-to-detect query containment situations arising from simple filter adjustments[2]; other cases, such as modifying a black-box classification/extraction operator, require the user to supply a migration plan. Furthermore, some migration plans are extremely efficient compared to re-processing from scratch, whereas others are less advantageous, or may even be worse depending on the available access methods. Lastly, some workflow changes affect the schema of data flowing to subsequent operations (e.g. a downstream workflow component that associates reviews with restaurants), requiring synchronization of logic upgrades across multiple workflow components.

We set aside for now the issues of generating migration plans and performing (cost-based) plan selection. The first step is to develop a framework for workflows that evolve incrementally in two dimensions (data and logic), and can accommodate a broad range of automatically- or manually-constructed migration plans. There are several challenges in developing such a framework, including:

- **Accommodate dynamic data flow structure.** A migration plan for a given workflow step may have dif-

[2]In general, this problem is closely related to the one of *answering queries using views* [14].

ferent data flow structure than normal processing. For example the migration plan may read data from the output channel. Hence, unlike in traditional systems the data flow structure of the workflow is not static, and the framework must accommodate this dynamicity.

- **Permit flexible workflow scheduling.** Some workflow elements process data in an intentionally delayed fashion, due to resource contention and/or relaxed application semantics. For example, arrival of newly-crawled web pages may trigger immediate news extraction, but the less-critical restaurant extractor might only be run occasionally on large input batches. Workflow logic updates should not interfere with this careful scheduling by forcing all data to be "flushed" through the workflow to achieve a synchronization barrier between data evolution and logic evolution. Instead, the framework should accommodate arbitrary scheduling in the presence of interleaved data and logic changes, and still offer clean semantics.
- **Coordinate schema changes across the workflow.** Some workflow logic changes alter the schema of intermediate data products, in turn affecting downstream logic. To avoid "breaking" a downstream operator that expects a certain schema but receives a different one, the framework must support synchronization of logic upgrades in multiple parts of the workflow. Again, this support must not impose restrictions on workflow scheduling by flushing and quiescing the workflow prior to a logic upgrade.

This paper addresses these challenges by proposing a formal framework for workflows with evolving data and logic, with incremental processing in both dimensions and flexible scheduling. The framework is implemented in a prototype workflow system on top of the Pig/Hadoop [1, 3] data processing environment. Experiments on real data show the benefits of incremental data migration as workflow logic evolves.

1.5 Outline

The remainder of this paper is organized as follows. Section 2 gives our model of data processing workflows whose processing logic evolves incrementally. Section 3 extends the model to handle changes in the workflow structure and data schema. Several topics that are important but largely orthogonal to the workflow evolution model (scheduling, provenance and space management) are discussed afterward in Section 4. Section 5 introduces our prototype system, and Section 6 describes experiments run using the prototype. Avenues for future work are discussed in Section 7.

2. CORE MODEL

Rather than choosing a specific workflow model from one of the data-intensive workflow contexts mentioned in Section 1.1 (scientific workflow, ETL, information extraction, map-reduce workflows), we use a generic model intended to capture the key elements common to these environments:

A *workflow* is a directed acyclic graph, with two types of vertices: *processing tasks* and *data channels* (tasks and channels, for short). A task is a data processing step, e.g. restaurant extraction; a channel is a data container, e.g. extracted restaurant records. Edges connect tasks to channels; no edge goes from a task to a task or from a channel to a channel.

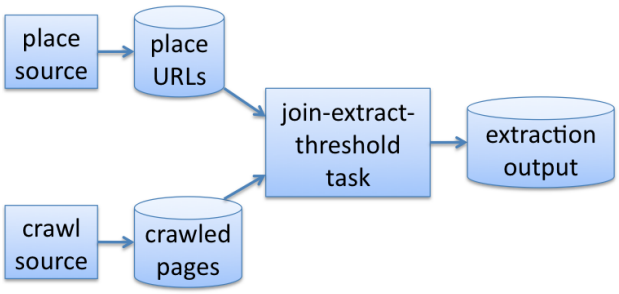

Figure 2: Formal workflow.

Figure 2 shows an example of a workflow fragment with tasks drawn as rectangles and channels drawn as cylinders. Figure 2 is based on Figure 1 but drawn in accordance with our formal model. Note that we have elected to group the join, extract and threshold steps into a single task. The choice of task boundaries is important, because tasks serve as the unit of scheduling (tasks may be invoked at different times and with different frequencies), checkpointing and sharing (task outputs are materialized on data channels).

As the reader has probably noticed in Figure 2, to simplify the formalism data sources are modeled as special *source tasks* that have no input channels—they inject data into the workflow (e.g. web pages from a crawl). Each channel is written to by exactly one task; if a channel's writer is a source task, then we call that channel a *source channel*. Channels that are not read by any tasks are called *output channels*—they constitute the workflow's data product.

2.1 Data Versioning

A channel's data is divided into *blocks*, each of which contains a set of data records or record transformations that have been generated by a single task invocation. Blocks may vary in size from kilobytes to terabytes. For the purpose of workflow management, a block is an atomic unit of data. Blocks also constitute atomic units of processing: a task invocation consumes zero or more input blocks and processes them in their entirety; partial processing of blocks is not permitted.

There are two types of blocks: *base blocks* and *delta blocks*. A base block contains a complete snapshot of data on a channel as of some point in time. A delta block contains instructions to transform a base block into a new base block, e.g. by adding, updating or deleting records or columns. Delta blocks are used for incremental processing; for now let us focus on non-incremental processing and base blocks.

Each base block on channel C is identified by a *version vector* $V = [v_1, v_2, \ldots, v_n]$; the block is called $B(V)$. The version vector length n equals the number of tasks that are ancestors of C in the workflow graph. Hence, for source channels $n = 1$; for the output channel in Figure 2, $n = 3$. As we explain below in Section 2.2, version vectors encode the coarse-grained data- and process-provenance of data blocks.

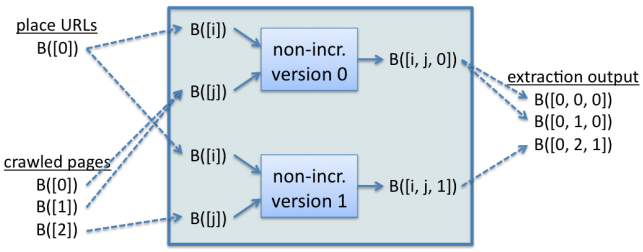

Figure 3: Non-incremental processing.

2.2 Process Versioning

A task contains a collection of *task instances*, which represent alternative ways to process the data. The alternatives may correspond to non-incremental vs. incremental processing, or may constitute evolving task logic.

Again restricting our attention for now to non-incremental processing, task instances correspond to new versions of the (non-incremental) task logic, and instances of a given task are assigned monotonically increasing version numbers. When a task instance is invoked it consumes one base block from each input channel, and emits a base block to the output channel. The version vector of the output block equals the concatenated version vectors of the input blocks, augmented with the version number of the task instance.

The user may add new instances to a task at any time. Arrival of a new task instance does not affect ongoing processing of input block(s), because block processing is atomic. Of course, subsequent input blocks might be processed using the new task instance, depending on the scheduling decision (Section 4.1).

Under normal circumstances,[3] task instances are not removed.

2.3 Example: Non-Incremental Processing

Figure 3 shows a close-up view of the join-extract-threshold task from Figure 2, with two task instances: instance 0 having the initial task logic, and instance 1 with upgraded logic registered at a later time. The place URLs input channel contains a single snapshot $B([0])$, and the crawled pages channel contains three successive snapshots, $B([0])$, $B([1])$ and $B([2])$, representing a series of complete ("from scratch") crawls. The figure shows the final task and data configurations resulting from the following series of events:

- Initial $B([0])$ input blocks arrive on both input channels.
- Task instance 0 is invoked on place URLs block $B([0])$ and crawled pages block $B([0])$, to produce output block $B([0, 0, 0])$.
- Crawled pages block $B([1])$ arrives.
- Task instance 0 is invoked again, this time on place URLs block $B([0])$ and the new crawled pages block $B([1])$, to produce output block $B([0, 1, 0])$. The second entry in the version vector indicates that $B([0, 1, 0])$ contains data from more up-to-date crawl snapshot than $B([0, 0, 0])$.

[3]Detection of a bug would require removing a task instance, as well as removing or correcting "polluted" downstream data using provenance metadata (Section 4.3).

- The user upgrades the task logic by instituting a higher confidence filter threshold, yielding task instance 1.
- Crawled pages block $B([2])$ arrives.
- Task instance 1 is invoked on place URLs block $B([0])$ and the latest crawled pages block $B([2])$, to produce output block $B([0, 2, 1])$. As the second and third version vector entries indicate, $B([0, 2, 1])$ reflects a more recent crawl snapshot as well as more recent task logic (new confidence threshold), relative to $B([0, 1, 0])$.

2.4 Incremental Processing and Migration

Incremental processing deals with delta blocks. A delta block is denoted $\Delta(V_1 \rightarrow V_2)$, where $V_1 \prec V_2$.[4] The block $\Delta(V_1 \rightarrow V_2)$ contains instructions and data for transforming $B(V_1)$ into $B(V_2)$. The process of applying a delta block to a base block, to form a new base block, is called *merging*, written $M(B(V_1), \Delta(V_1 \rightarrow V_2)) = B(V_2)$.[5] The reverse transformation, whereby two base blocks are compared to create a delta block that transforms one to the other, is called *diffing*: $D(B(V_1), B(V_2)) = \Delta(V_1 \rightarrow V_2)$.

A task instance that produces one or more delta blocks is termed *incremental*. One that produces only base blocks is *non-incremental*. An incremental task instance I is *correct* iff for all delta blocks $\Delta(V_1 \rightarrow V_2)$ that I emits to channel C, if we were to generate base blocks $B(V_1)$ and $B(V_2)$ for channel C using non-incremental instances it would be the case that $M(B(V_1), \Delta(V_1 \rightarrow V_2)) = B(V_2)$.

When we include incremental processing options, task instances are no longer identified by a single version number. Instead, a task instance has a *signature* that specifies:

- the number of base and delta blocks to read from each channel sharing an edge with the task,
- zero or more constraints on the version vectors of the blocks to be read,
- the type of block to be written (base or delta), and
- output block version vector(s) as a function of the input block version vector(s).

2.5 Example: Incremental Processing

Figure 4 adds incremental task variants to our example join-extract-threshold task from Figure 3, and illustrates the following sequence of events (the first two of which are the same as in Figure 3):

- Initial $B([0])$ input blocks arrive on both input channels.
- The non-incremental variant of task version 0 is invoked on place URLs block $B([0])$ and crawled pages block $B([0])$, to produce output block $B([0, 0, 0])$.
- Crawled pages block $\Delta([0] \rightarrow [1])$ arrives from an *incremental crawler* [8] that has found additional pages,

[4]The partial order $\prec$ is defined on a pair of equal-length version vectors $V_1 = [v_{1,1}, v_{1,2}, \ldots, v_{1,n}]$ and $V_2 = [v_{2,1}, v_{2,2}, \ldots, v_{2,n}]$, as follows: $V_1 \prec V_2$ iff for all $1 \leq i \leq n$, $v_{1,i} \leq v_{2,i}$ and for some $1 \leq j \leq n$, $v_{1,j} < v_{2,j}$.

[5]In general, merging is performed over one base block and $n \geq 1$ delta blocks: $M(B(V_1), \Delta(V_1 \rightarrow V_2), \Delta(V_2 \rightarrow V_3), \ldots, \Delta(V_n \rightarrow V_{n+1})) = B(V_{n+1})$.

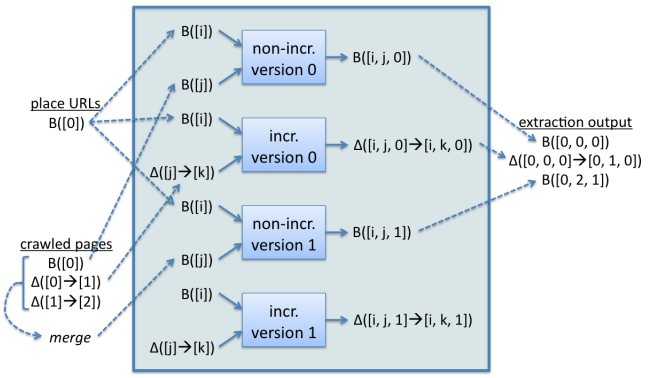

Figure 4: Incremental processing, without migration plan.

and has re-crawled some previously-crawled pages to find changed content and "dead" pages (404 errors). The delta block contains upserts (insertions and replacements) and deletions, using URL as a key.

- The incremental variant of task version 0 is invoked on place URLs block $B([0])$ and crawled pages delta block $\Delta([0] \rightarrow [1])$, producing an output delta block $\Delta([0,0,0] \rightarrow [0,1,0])$ that inserts, replaces and deletes[6] restaurant records using URL as a key.
- The user upgrades the task to version 1, by supplying non-incremental and incremental variants of the new task logic.
- Crawled pages block $\Delta([1] \rightarrow [2])$ arrives from the crawler.
- The non-incremental variant of task version 1 is invoked on place URLs block $B([0])$ and the on-the-fly merge of all three crawled pages blocks, $M(M(B([0]), \Delta([0] \rightarrow [1])), \Delta([1] \rightarrow [2]))$, to produce output block $B([0,2,1])$.[7]

2.6 Example: Incremental Migration

Figure 5 adds another task variant to our running example, which performs incremental migration from version 0 to 1 by filtering the old output based on the new, higher, confidence threshold. In our model, a migration task instance is no different from any other task instance. (Therefore incremental migration instances are subject to the same correctness criterion as regular incremental task instances (Section 2.4), i.e. the emitted delta blocks must correctly transform the old output data to the new output data.) Of course, migration instances typically read from the output channel, whereas most non-migration instances read solely from input channels.

[6]In this incremental processing strategy, records that do not pass the confidence filter are sent to the output as deletions, and deletion is treated as an idempotent operation in the output channel merge operation. (The overhead of these conservative deletions can be eliminated via a more complex incremental task variant that reads the URL column of the prior output. While our model supports such variants, exploring the tradeoffs between different incremental processing strategies is outside the scope of this paper.)

[7]If a consumer of the extraction output channel wishes to consume data in the form of deltas, the delta block $\Delta([0,1,0] \rightarrow [0,2,1])$ can be generated from the existing output blocks using diffing (combined with merging): $D(M(B([0,0,0]), \Delta([0,0,0] \rightarrow [0,1,0])), B([0,2,1])) = \Delta([0,1,0] \rightarrow [0,2,1])$.

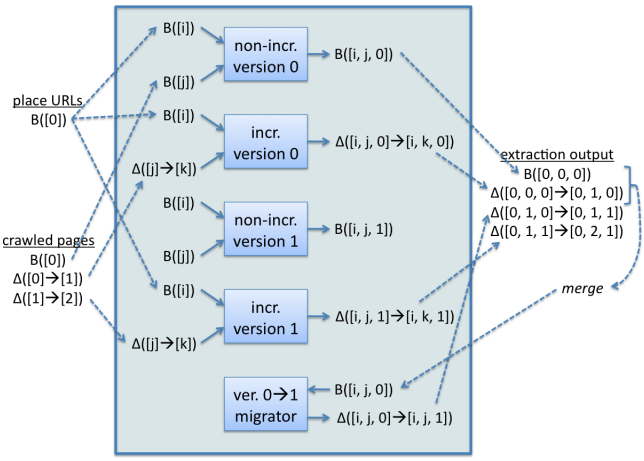

Figure 5: Incremental processing with migration plan.

In Figure 5 the migration instance has been invoked on version 0 output data $M(B([0,0,0]), \Delta([0,0,0] \rightarrow [0,1,0]))$ to produce output delta block $\Delta([0,1,0] \rightarrow [0,1,1])$, which deletes records with confidence values that are below the version 1 threshold.

2.7 Discussion of Incremental Migration

As stated in Section 2.6, migration task instances are not treated as special cases in the model—they behave just like other task instances. Our intent is to constrain the structure of migration plans as little as possible, in view of accommodating a wide variety of incremental migration strategies that might be devised to handle various kinds of workflow logic changes.

Of course, the potential for real-world impact of our approach hinges on the ability to devise efficient migration plans for common workflow evolution scenarios. Fortunately there appears to be some "low-hanging fruit" in terms of minor, transparent logic changes that occur frequently in practice, e.g.:

- Changes to, additions of, and removal of filter conditions, such that there is a query containment relationship between the old and new output data sets that implies compact delta blocks for inserting or removing the affected output records.
- Changes to column projection lists, which can be handled efficiently via delta blocks that add/drop columns, especially if the underlying storage system uses columnar storage.

We also anticipate that in many cases, changes to black-box workflow steps (e.g. custom information extraction functions) can be paired with manually-written incremental migration plans, e.g. if the new extraction function just extracts an additional attribute, or extracts an existing attribute differently but leaves the other attributes unaffected. (Note that manually-supplied migration plans facilitate incremental processing just as well as automatically-generated ones—the only difference is that the incremental data processing algorithm has been written by hand.)

Lastly, as we show next in Section 3 our model can be extended to accommodate addition and removal of data sources, which enables efficient incremental migration in common scenarios such as:

- Adding an input data source that is "unioned" with existing sources (e.g. extending the scope of movie data extraction from { IMDB, Yahoo! Movies } to { IMDB, Yahoo! Movies, Netflix }).
- Inserting a foreign-key join to create an additional derived attribute (e.g. "geo-tagging" to convert IP addresses to zip codes via a foreign-key join with an IP-to-zipcode lookup table from a new source).

3. MODEL EXTENSIONS

Having presented our basic model, we introduce two important extensions: accommodating workflow structure evolution and schema evolution. We first give brief, informal overviews of each topic and our approach (Section 3.1), and then supply the details for readers who are interested (Sections 3.2–3.3).

3.1 Overview

3.1.1 Workflow Structure Evolution

So far we have focused on evolution of an existing workflow task's logic. The structure of the workflow may also evolve (adding and removing task nodes; connecting and disconnecting channels from tasks). It is worth noting that addition of tasks is not common, because it implies creating an additional channel which adds materialization overhead—instead, new processing steps (e.g. a spam filter) are generally incorporated into existing tasks. That said, our framework does handle adding new tasks, as well as the other structural evolution scenarios mentioned.

Our general strategy is to reduce workflow structure evolution to task logic evolution, and thereby leverage the task logic evolution machinery we have already developed (Section 2). For example, we can pretend that a new task has always existed, but until now it only performed no-ops. Similarly, we can pretend that a newly-created channel has always existed and been attached to a particular task, but thus far the task has only read the initial, empty data snapshot. The details of how we extend our framework to handle structural evolution are given in Section 3.2.

3.1.2 Schema Evolution

As discussed in Section 1.4, workflow logic upgrades may alter the schema of derived data. Some modeling and machinery is required to coordinate schema evolution across multiple workflow tasks. At a minimum, we need to ensure that downstream tasks do not "break" when upstream tasks alter their output schemas.

Our framework for managing schemas and synchronizing task upgrades is described in Section 3.3. It has two attractive properties: (1) The user does not need to submit a set of task upgrades as some sort of transaction; instead she can upgrade the tasks one-by-one without fear of schema mismatch errors; (2) The system does not need to "flush" old data through the workflow prior to installing a task upgrade that induces a schema change; data conforming to the old and new schemas can be present at the same time without causing problems.

3.2 Workflow Structure Evolution

To help us deal with workflow structural changes, we introduce two special version numbers, s (for start) and e (for end), which act as extrema in the version number ordering: $s < 0 < 1 < 2 < \ldots < e$. We will use these special version numbers to treat structural changes as if they were task logic changes (e.g. pretend that a new task has always existed in the form of a no-op as version s). This approach allows us to leverage the task logic evolution machinery we have already developed (Section 2).

The remainder of this subsection explains how each structural evolution case is handled.

3.2.1 Inserting a Task

When a new task is inserted into a workflow, the channel at the insertion point is split into two channels (becoming the input and output channels of the new task). Both channels contain copies of the blocks present on the original channel.[8] Then, the version vectors of all blocks downstream of the new task are augmented to make it appear as though the new task had always been there in the form of a no-op instance, with version s—i.e. an s is added to each version vector at the appropriate position. After that, the user may begin registering instances of the new task, starting at version 0 (perhaps including a migration plan from version s to 0). Other than the brief pause to modify downstream version vectors, workflow processing proceeds in the normal fashion as described in Section 2.

In our running example, suppose a web page spam filtering task is inserted between the crawl source and the main join-extract-threshold task. (Perhaps the user chose to perform the spam filtering in a separate task, rather than at the beginning of the existing join-extract-threshold task, to enable its output to be multiplexed to multiple consumers in the future.) Consider the execution sequence illustrated in Figure 4, and suppose that at the time of the task insertion only the first block on each channel is present. Insertion of the new task causes a new channel to be created at the left input to the join-extract-threshold task, with a copy of the crawled pages block $B([0])$ renamed to $B([0, s])$. The extraction output block $B([0, 0, 0])$ is renamed to $B([0, 0, s, 0])$. Assuming a migration plan is provided for the new spam filter task, when it is invoked it reads block $B([0, s])$ from its output channel and produces a delta block $\Delta([0, s] \rightarrow [0, 0])$ that eliminates spam pages from $B([0, s])$. This delta block can be consumed by the incremental variant of the join-extract-threshold task, resulting in an output block $\Delta([0, 0, s, 0] \rightarrow [0, 0, 0, 0])$. From there, the workflow behaves as if the spam filter had always been present. Notice that the spam filter was incorporated in an incremental fashion, i.e. the non-incremental variant of the join-extract-threshold task did not have to be invoked.

[8]Since blocks are read-only, copy-by-reference may be used, i.e. both channels "point to" the blocks, although they must keep separate version vector metadata.

Inserting a multi-input task is accomplished by first inserting a single-input variant (as just described), and then attaching additional inputs (as described in Section 3.2.3).

3.2.2 Removing a Task

A task can only be removed from a workflow if it has at most one input. A multi-input task can be retired by first reducing it to a single-input task by detaching inputs (see Section 3.2.4), and then removing the task.

The system handles a task removal request by registering a no-op instance of the to-be-removed task with version e (the user is given the opportunity to supply a migration plan from the last version to e). To avoid unnecessary processing and space overhead, the no-op instance uses copy-by-reference to transfer input blocks to the output, as described in Section 3.2.1. The no-op task may remain in existence for some time (masked from the user by the system), which is of little concern since it incurs almost no processing and space overhead. Eventually, when no version numbers from that task other than e still linger in the workflow (i.e. they have been garbage collected; see Section A.2), the task and its output channel are eliminated, and its input channel is connected to the downstream task. At that point, the e version vector entries are removed from all block metadata.

3.2.3 Attaching a Task Input

One may wish to add an input to an existing task. For example, restaurant address extraction may be made more accurate by referencing a city name data set, supplied on a new input channel. This scenario is handled by augmenting downstream version vectors with s entries at the appropriate positions. The user is then free to register instances of the task that read from the new input, including migration plans.

3.2.4 Detaching a Task Input

To detach a task input I, the user first registers task instance(s) that ignore I (i.e. they do not request any blocks from I). The system automatically pads version vectors produced by such instances with e entries in the position(s) corresponding to I. Eventually, when no downstream blocks remain that do not have e entries in those position(s), the e entries are removed and I is detached from the task. As in the task removal case (Section 3.2.2), in the mean time the system may give the user the illusion that I has already been detached from the task, by masking I and the e entries.

3.3 Schema Evolution

To detect and react to schema changes resulting from workflow logic changes, the system maintains a global *schema function* $\mathcal{S}(C, V)$ that associates a schema S with version vector V on channel C. The schema function is constructed from schema specifications supplied by the tasks. Source tasks declare the schema of every version (by default, the schema of version $i+1$ is the same as that of version i). Non-source tasks supply *schema transformation* functions, which give the output schema as a function of the input channel schema(s).

As an aside, the output schema need not be closely related to the input schema—in such cases the schema transformation function can ignore the input schema and simply return a constant representing the output schema. The reason for encoding schemas as transformations, rather than representing each task's schema independently, is to minimize the schema encoding work for operations like filtering and projection, whose output schema is identical or closely related to the input schema.

Formally, each task T maintains a schema transformation function $\mathcal{T}_T(v)$ that consumes a version number v of task T and returns a function $f : (S_1, S_2, \ldots, S_k) \to S$ where $S_1, S_2, \ldots, S_{I(T)}$ are the schemas of the $I(T)$ input channels and S is the output schema. If T is a source task, $I(T) = 0$ and $\mathcal{T}_T(v)$ returns a constant function; in other words the schema "transformation" simply emits the schema of version v of the source data.

The global schema function $\mathcal{S}(C, V)$ is defined recursively over the task schema transformation functions:

$$\mathcal{S}(C, V) = \left[\mathcal{T}_{W(C)}(V_{|V|})\right](\mathcal{S}(C\langle 1\rangle, V\langle 1\rangle), \mathcal{S}(C\langle 2\rangle, V\langle 2\rangle), \ldots)$$

where V_i gives the ith element of V, $W(C)$ is the task that writes to channel C, $C\langle i\rangle$ gives the ith input channel of $W(C)$, and $V\langle i\rangle$ gives the subrange of V that comes from $C\langle i\rangle$.

This model is agnostic to the schema representation and schema transformation algebra. The schema may include logical elements only (i.e. field names and types, and integrity constraints such as uniqueness and functional dependencies), or it may also incorporate physical properties like ordering and partitioning. Tracking physical properties enables optimizations whereby a downstream task algorithmically exploits a physical property of the incoming data. Designing a schema transformation algebra is straightforward; it would have operations for adding and removing fields, integrity constraints and physical properties.

With a simple representation in which a schema just consists of a set of field names, with no data types or physical properties, the schema transformation functions for the tasks in our running example from Figure 2 are:

- $\mathcal{T}_{place\ source}(0) = f : () \to \{url\}$
- $\mathcal{T}_{crawl\ source}(0 \ldots 2) = f : () \to \{url, content\}$
- $\mathcal{T}_{join-extract-threshold}(0 \ldots 1) = f : (S_1, S_2) \to (S_2 \setminus \{content\}) \cup \{name, address, hours, confidence\}$

and the global schema function evaluates to:

- $\mathcal{S}(place\ URLs, [0]) = \{url\}$
- $\mathcal{S}(crawl\ source, [0 \ldots 2]) = \{url, content\}$
- $\mathcal{S}(extraction\ output, [0, 0 \ldots 2, 0 \ldots 1]) = \{url, name, address, hours, confidence\}$.

If, starting with version 3, the crawl source were to add a new attribute to indicate the time at which the content was crawled, we would have:

- $\mathcal{T}_{crawl\ source}(3) = f : () \to \{url, time, content\}$

- $\mathcal{S}(\textit{crawl source}, [3]) = \{\textit{url}, \textit{time}, \textit{content}\}$
- $\mathcal{S}(\textit{extraction output}, [0, 3, 0 \ldots 1]) =$ $\{\textit{url}, \textit{time}, \textit{name}, \textit{address}, \textit{hours}, \textit{confidence}\}$.

3.3.1 Synchronized Upgrades and Schema Constraints

Continuing our example, suppose the user does not wish to include crawl time in the extraction output, so she supplies a new version (version 2) of the join-extract-threshold task that projects out that attribute, which has schema transformation function $\mathcal{T}_{join-extract-threshold}(2) = f : (S_1, S_2) \rightarrow (S_2 \setminus \{\textit{time}, \textit{content}\}) \cup \{\textit{name}, \textit{address}, \textit{hours}, \textit{confidence}\}$. Clearly, we must ensure that version 2 of the join-extract-threshold task instance is not invoked on data that predates version 3 of the crawl source, because the projection step would throw an error.

In our framework, synchronization of multiple task upgrades is achieved via *schema constraints*, whereby task instances specify which logical and physical schema characteristics are relied on. In our example above, version 2 of the join-extract-threshold task would include crawl time among the schema constraints of its second input. The system will automatically switch to version 2 at the appropriate time, i.e. after any enqueued data from prior versions of the crawl source has been processed.

Another scenario is when the user who programs a given task is not aware that an upstream schema change has occurred. If the system detects a situation in which data is "stuck" in front of a task because no instance of the task accepts the data's schema, an alert is generated prompting the user to upgrade the task. In fact, using the schema function, the system can anticipate and generate an alert for this situation as soon as an upstream task is upgraded such that a schema incompatibility with a downstream task is induced. An alternative rule would be to reject the upstream upgrade if there is no downstream instance that can accept the new schema. (These choices are reminiscent of alternatives for handling data deletions in the presence of referential integrity constraints.)

Schema transformations and constraints may be supplied by hand or inferred from the task logic. If the task logic is written in a high-level language like SQL or Pig Latin, this inference should be fairly straightforward. However, user-defined functions (UDFs) may require manual schema specification, and although their schema constraints can be inferred, the inferred constraints might be conservative (e.g. field X is input to the UDF but not actually used by the UDF, perhaps for historical reasons or because the UDF's input was set to "$*$" for simplicity). Physical constraints (e.g. merge join requires inputs to arrive sorted on the join keys) may be more difficult to infer automatically.

4. SCHEDULING, PROVENANCE AND SPACE MANAGEMENT

This section covers topics that are not central to the framework but are nonetheless very important in practice. Since these are not core topics to the paper, we give only a high-level overview here; some details are found in Appendix A.

4.1 Task Scheduling

One of our stated goals (Section 1.4) is to permit workflow task invocations to be scheduled in a flexible manner. The only constraint we impose is *version monotonicity*, defined with respect to each channel's *frontier*. Channel C's frontier $\mathcal{F}(C)$ is the set of maximal[9] version vectors found in a base block, or chained from a base block through a series of deltas. Under the version monotonicity constraint, no task invocation may emit a block $B(V)$ or $\Delta(V_0 \rightarrow V)$ to channel C if for some $F \in \mathcal{F}(C)$, $V \prec F$.

Many conceivable scheduling objectives and algorithms exist, including ones that trade off latency and consistency, and ones that defer expensive task upgrades until a time at which system load is low. This paper does not introduce any special scheduling methods. The simple scheduling heuristic we implemented in our prototype is described in Section 5.

4.2 Data Compaction

As delta blocks accumulate on a channel, it is generally desirable to *compact* the channel's data by running a merge operation (Section 2.4) and materializing the result in a new base block. Compaction reduces the on-the-fly merge overhead for subsequent non-incremental task executions. Moreover, if the delta blocks include updates and/or deletions of existing data items, the compacted representation saves space. Of course, the space reduction is not realized until the old base and delta blocks are removed—our next topic.

4.3 Data Provenance and Garbage Collection

Over time, as data accumulates on a channel and old data is compacted (Section 4.2), it becomes necessary to remove old blocks; this process is called *garbage collection*. The formal rules for garbage collection, along with an example, are presented in Appendix A. The general idea is that a block is eligible for garbage collection if it is redundant with another block (e.g. following a compaction event, the old base and deltas are redundant with the new compacted base), and is not needed for future incremental processing by a downstream task (redundant deltas might still be kept if a downstream incremental task has not yet read them).

Another constraint imposed on garbage collection is that it cannot leave dangling *data provenance* references. As with many workflow environments, our framework tracks coarse-grained data provenance, in the form of directional references between pairs of blocks. In the context of incremental processing, there are two types of provenance references: physical references and logical references. Physical provenance traces the actual sequence of intermediate data products that led to a particular piece of data, whereas logical provenance traces the equivalent non-incremental path (formal definitions and examples are found in Appendix A).

Physical provenance is useful for debugging errors in the incremental processing logic, and for understanding the way the workflow has been scheduled. Logical provenance is much simpler in that it does not require understanding incremental processing semantics, and it also enables much more aggressive garbage collection, as we demonstrate in

[9]Maximality is defined with respect to the partial order $\prec$ defined in Section 2.4.

Section 6.5. We expect that real systems would offer two modes of operation:

- **Test mode:** Test the workflow on a small amount of data, with garbage collection disabled, to debug the incremental processing logic.
- **Production mode:** Run the workflow on large data in a production setting, with garbage collection constrained only by logical provenance.

5. PROTOTYPE WORKFLOW MANAGER

Our prototype workflow management system is implemented in Java, on top of the Pig/Hadoop [1, 3] data processing environment. The workflow system manages metadata about tasks, channels, blocks, version vectors, logical and physical provenance, and schemas. Our prototype uses a simple schema model (a set of field name/type pairs) and schema transformation algebra (add field and drop field). To link the metadata to the underlying Hadoop system, the workflow manager maintains a mapping from data blocks to files in the Hadoop file system (HDFS), along with each block's storage format (e.g. binary or text, compressed or uncompressed).

The workflow manager permits new task instances, including migration plans, to be registered at any time (workflow evolution). Task instances take the form of Pig Latin scripts [20], but rather than referencing specific input and output HDFS files, channel name placeholders are used. When the workflow manager invokes a task instance, it substitutes these placeholders for specific block file names or, in the case of on-the-fly merging, Pig Latin expressions that merge several block files.

The system follows a simple task scheduling heuristic: It visits workflow tasks in a round-robin fashion, and considers task instances that are eligible for execution (i.e. all required input blocks are present, or can be created via on-the-fly merging). The highest priority eligible instance is selected for execution, with migration plan instances given top priority, followed by incremental instances, and finally non-incremental instances. If there is a tie (e.g. no migration instances and multiple incremental instances), then the one that advances the frontier (Section 4.1) of the task's output channel by the greatest amount (using L1 distance) is chosen.

Data compaction is scheduled at fixed intervals, and the garbage collector is invoked after each round of compaction. Three garbage collection (GC) options are supported: (1) no GC, (2) GC constrained by physical provenance, and (3) GC constrained by logical provenance.

6. EXPERIMENTS

We built a prototype workflow management system on top of the Pig/Hadoop [1, 3] data processing environment, and conducted a series of experiments. Our experiments are designed to quantify the benefit of incremental data migration when workflows evolve, and to measure space footprint growth under various compaction and garbage collection strategies. They use two realistic workflows over real web data, running on an eight-node Hadoop cluster.

Before describing our experiments and results, we give the details of our data, workflows and execution environment.

6.1 Data

Our workflows use two input data sets:

- `Pages`: A 9 GB web page sample containing metadata about ten million URLs, such as their file type (HTML, PDF, etc.), language (English, French, etc.), sets of outgoing and incoming hyperlinks, and a quality score $q \in [0, 1]$ (computed from URL features and other evidence). To simulate incremental arrival from a crawler, we divided this data set into ten roughly equal-sized chunks, each stored in a separate HDFS file.
- `Sites`: A 43 MB site-level data set that contains one record per web site that appears one or more times in `Pages`, with each site's quality score (computed using link analysis and other methods).

6.2 Workflows

Our experiments are over two workflows, each having a single non-source task:

- `Workflow 1` filters incrementally-arriving `Pages` delta blocks by quality ($q > T$, for some threshold T), and performs a join with `Sites` to pick up the site-level quality score of each page that survives the page-level quality filter. The page-level quality threshold is initially set to $T = 0.6$, and later lowered to $T = 0.5$. When the threshold is lowered, a migration plan is supplied that reads the old `Pages` data, filters by $0.5 < q \leq 0.6$, and joins with `Sites` to produce an output delta block.[10]
- `Workflow 2` groups `Pages` deltas by site, finds the number of distinct incoming hyperlinks (c) to each site, filters by hyperlink count ($c > T$), and emits the site/count pairs.[11] The link count threshold is initially set to $T = 200$, and later adjusted to either $T = 100$ or $T = 400$ (our experiments cover both scenarios). In the case of raising the threshold, the migration plan simply filters the output data by the new, higher, threshold. In the case of lowering the threshold, the migration plan re-runs the task on the merged view of the entire accumulated input data, but inserts an anti-join with the old output data to avoid re-processing sites whose count is above the old threshold of 200; the result is an output delta containing sites with $100 < c \leq 200$.

6.3 Execution Environment

In our experiments we ran our prototype workflow manager (Section 5) on a machine with a 2.53 GHz dual-core CPU and 4 GB of RAM. The workflow manager connects over TCP to a Hadoop cluster with one master node (running the

[10]The case of raising the threshold results in a migration strategy that is identical except that the output delta tuples are marked as "negative tuples," so we did not study it as a separate experiment.

[11]This efficient incremental strategy for distinct counting assumes that the blocks from the crawler do not overlap by site—e.g. the crawler explores one site at a time (or a small set of sites at a time to spread out requests for politeness), which is the pattern Googlebot apparently follows [23], and emits blocks at site boundaries.

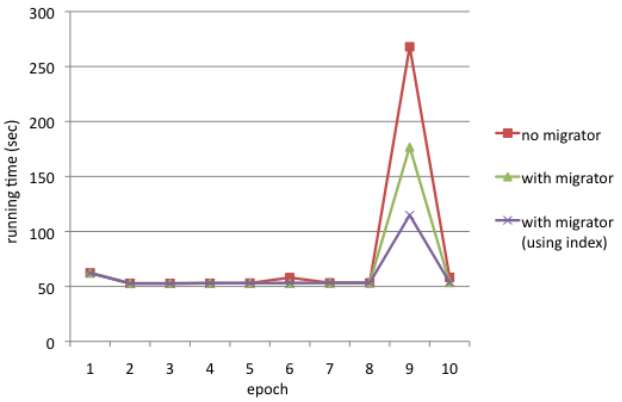

Figure 6: Running time (`Workflow 1`).

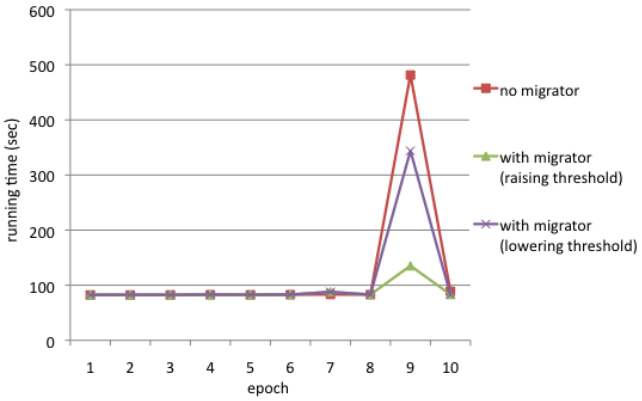

Figure 7: Running time (`Workflow 2`).

map-reduce JobTracker and HDFS NameNode) and eight slave nodes (each running a map-reduce TaskTracker and an HDFS DataNode). Each of the Hadoop cluster nodes have 2.13 GHz dual-core CPUs and 4 GB of RAM, and run the Linux operating system.

6.4 Running Time Measurements

Figures 6 and 7 show the running times of `Workflows 1` and 2, respectively, with compaction and garbage collection disabled to isolate the task running times. In each graph the horizontal axis plots the workflow epoch (processing associated with each of the ten incoming `Pages` blocks). The vertical axis of plots the running time for the work performed in a given epoch (averaged over five runs, with outliers removed).

Figure 6 shows three cases: one in which the migration plan is disabled, one with the migration plan is enabled, and one with the migration plan enabled and the `Pages` data indexed by quality score. (We simulated the index in Hadoop by dividing the data into ten quality-score buckets, and storing each bucket's data in a separate file.) Figure 7 also shows three cases, but they are different: the first case has the migration plan disabled, the second case has a migration plan for raising the count threshold, and the third case has a migration plan for lowering the count threshold.

The per-epoch processing time is nearly constant except in the ninth epoch, when the workflow evolution occurs. Without migration, there is a major hiccup to re-process all accumulated input data from scratch. In our scenarios the hiccup amounts to a five-fold processing time increase for that epoch. In general there is no bound to the data re-processing overhead: the more time has elapsed the more accumulated input data would need to be re-processed.

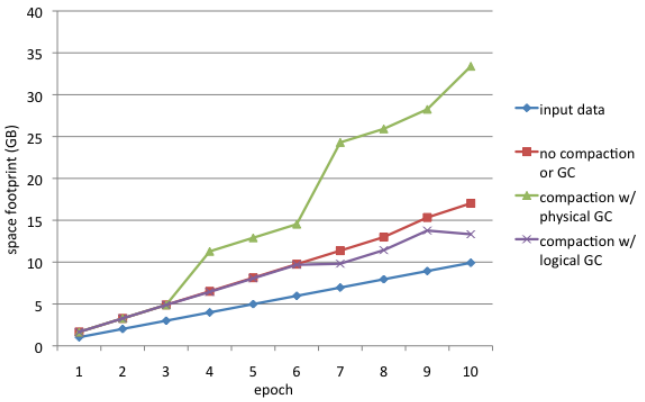

Figure 8: Space footprint.

As Figures 6 and 7 show, migration plans greatly reduce the overhead of adapting to an evolving workflow. Of course, the magnitude of reduction depends on the way in which the workflow has evolved (as shown in Figure 7) and the physical database design (Figure 6).

In some cases the asymptotic running time is also reduced compared with re-processing from scratch. For example, for raising the threshold in `Workflow 2`, the running time becomes a function of the output data size rather than the input size, and the output is much smaller than the input.

6.5 Space Footprint Measurements

To create a situation in which compaction reduces the data size, we artificially modified `Workflow 1` to allow merging (and thus compaction) of the output channel by site. Figure 8 shows the total space footprint (vertical axis) over time (horizontal axis). (`Workflow 2`'s output is very small and the space footprint is dominated by the input data, making it uninteresting in this experiment.) The lowest curve shows the size of the input data, which grows linearly. The other three curves show the total data size, under three scenarios: (1) no compaction or garbage collection (GC); (2) compaction with GC constrained by physical provenance; and (3) compaction with GC constrained by logical provenance.

As expected, constraining GC by physical provenance eliminates most garbage-collection opportunities, and leads to multiple redundant copies of the data following compaction. Constraining GC by logical provenance leads to much more sensible space footprint characteristics, with the combination of compaction and garbage collection actually saving space as time goes by.

7. SUMMARY AND FUTURE WORK

We have described a model and system for processing workflows incrementally as their data and logic both evolve. Our approach handles very general categories of workflow changes, including ones that impact the workflow graph structure

or data schema. Additionally it can handle a wide array of incremental processing strategies, including ones that reprocess old output data. Lastly, it does not interfere with workflow scheduling, e.g. it does not require workflow processing to be "flushed" or "paused" to accommodate a workflow logic upgrade.

By preserving scheduling flexibility, we leave the question of optimal scheduling, including how often to perform compaction, to be addressed in separate work. One can imagine a cost-based scheduler that aims to minimize latency or maximize throughput. Since migration plans are no different from regular workflow operations in our model, they should not present a major complication for scheduling.

Automatic migration plan generation was left as future work. There appears to be some low-hanging fruit in this area. For example, it should be fairly easy to generate migration plans for filter or projection logic changes using basic query-containment-style reasoning. In general, this problem is closely related to the one of *answering queries using views* [14], and techniques from that area are likely to be applicable.

Acknowledgments

We thank Phil Bohannon for helpful discussions about PSOX workflows.

8. REFERENCES

[1] Apache. Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`.

[2] Apache. Oozie: Hadoop workflow system. `http://yahoo.github.com/oozie/`.

[3] Apache. Pig: Dataflow programming environment for Hadoop. `http://pig.apache.org`.

[4] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Enabling interactive multiple-view visualizations. In *Proc. IEEE Visualization*, 2005.

[5] P. Bohannon, S. Merugu, C. Yu, V. Agarwal, P. DeRose, A. Iyer, A. Jain, V. Kakade, M. Muralidharan, R. Ramakrishnan, and W. Shen. Purple SOX extraction management system. *ACM SIGMOD Record*, 37(4):21–27, 2008.

[6] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *IEEE Trans. on Data and Knowledge Engineering*, 24(3):211–238, 1998.

[7] C.-H. Chang, M. Kayed, R. Girgis, and K. F. Shaalan. A survey of web information extraction systems. *IEEE Trans. on Knowledge and Data Engineering*, 18(10):1411–1428, 2006.

[8] J. Cho and H. Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *Proc. VLDB*, 2000.

[9] C. Date and H. Darwen. *Temporal Data and the Relational Model*. Morgan Kaufmann, 2002.

[10] C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proc. Conference on Organizational Computing Systems*, 1995.

[11] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *Proc. International Provenance and Annotation Workshop*, 2006.

[12] P. J. Guo and D. Engler. Towards practical incremental recomputation for scientists: An implementation for the Python language. In *Proc. USENIX Workshop on the Theory and Practice of Provenance*, 2010.

[13] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):5–20, 1995.

[14] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10:270–294, 2001.

[15] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, and L. Zhou. Comet: Batched stream processing for data intensive distributed computing. In *Proc. ACM Symposium on Cloud Computing (SOCC)*, 2010.

[16] T. Jorg and S. DeBloch. Towards generating ETL processes for incremental loading. In *Proc. 12th International Database Engineering and Applications Symposium (IDEAS)*, 2008.

[17] A. Koeller and E. A. Rundensteiner. Incremental maintenance of schema-restructuring views. In *Proc. EDBT*, 2002.

[18] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Stateful bulk processing for incremental algorithms. In *Proc. ACM Symposium on Cloud Computing (SOCC)*, 2010.

[19] B. Ludascher et al. Scientific process automation and workflow management. In *Scientific Data Management: Challenges, Technology, and Deployment*, chapter 13. Chapman & Hall/CRC, 2009.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, 2008.

[21] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *Proc. USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2008.

[22] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[23] `http://www.webmasterworld.com`.

APPENDIX

A. DATA PROVENANCE AND GARBAGE COLLECTION

A.1 Provenance

Physical and logical provenance are defined formally as follows:

- **Physical provenance:** If block X is part of block Y's physical provenance, that indicates that a task instance or compaction operation was executed with X as part of its input, and with Y as its output.
- **Logical provenance:** Logical provenance references only apply to base blocks. If base block B_1 is part of base block B_2's logical provenance, that indicates that there exists a non-incremental task instance that would produce B_2 if executed with B_1 as part of its input. Using the notation introduced in Section 3.3, the logical provenance of block $B(V)$ on channel C, which we'll denote $B_C(V)$, is $\{B_{C\langle 1\rangle}(V\langle 1\rangle), B_{C\langle 2\rangle}(V\langle 2\rangle), \ldots\}$.

As an example, consider the blocks shown in Figure 4, and let us use the channel abbreviations p = *place urls*, c = *crawled pages* and e = *extraction output*. The physical provenance of output block $B_e([0,2,1])$ is $\{B_p([0]), B_c([0]), \Delta_c([0] \rightarrow [1]), \Delta_c([1] \rightarrow [2])\}$, whereas its logical provenance is $\{B_p([0]), B_c([2])\}$. As a second example, if one were to compact output blocks $B_e([0,0,0])$ and $\Delta_e([0,0,0] \rightarrow [0,1,0])$ to produce $M(B_e([0,0,0]), \Delta_e([0,0,0] \rightarrow [0,1,0])) = B_e([0,1,0])$, that block's physical and logical provenance would be $\{B_e([0,0,0]), \Delta_e([0,0,0] \rightarrow [0,1,0])\}$ and $\{B_p([0]), B_c([1])\}$, respectively.

In a workflow with only non-incremental task variances, physical and logical provenance are the same. In an incremental workflow, it may often happen that a block referenced by another block's logical provenance does not exist. However in such cases it is always possible to construct the non-existent block via an on-the-fly merge of existing base and delta blocks (assuming that they have not been garbage-collected; see Section A.2).

A.2 Garbage Collection

A base or delta block b on channel C (i.e. $b = B_C(V)$ or $b = \Delta_C(V_0, V)$) is eligible for garbage collection iff all of the following conditions hold:

1. Block b is *subsumed* by another block on C, i.e. there exists a base block $B_C(V') \neq b$ such that $V \preceq V'$.
2. Based on the version monotonicity scheduling restriction (Section 4.1), it can be determined that no future task invocation would read b.
3. No other block b' has a provenance reference to b.

Condition 1 ensures that we only remove old or redundant data. Condition 2 ensures that we do not disrupt incremental processing: Say a task's incremental variant has processed $\Delta([0] \rightarrow [1])$ and the next invocation would process $\Delta([1] \rightarrow [2])$. We do not want the existence of a compacted base $B([2])$ to trigger garbage collection of $\Delta([1] \rightarrow [2])$ and therefore force the task to revert to non-incremental processing of $B([2])$.

Condition 3 avoids dangling provenance references. This condition can be enforced with respect to either physical or logical provenance. The logical-provenance enforcement option enables much more aggressive garbage collection, as illustrated in the following example and quantified empirically in Section 6.5.

In Figure 4 blocks $B_e([0,0,0])$ and $\Delta_e([0,0,0] \rightarrow [0,1,0])$ are eligible for garbage collection because they satisfy Condition 1, and Conditions 2 and 3 are not applicable because no tasks consume data from the extraction output channel. Suppose that the crawled pages channel is compacted, yielding block $M(B_c([0]), \Delta_c([0] \rightarrow [1]), \Delta_c([1] \rightarrow [2])) = B_c([2])$. If Condition 3 is enforced with respect to logical provenance, then blocks $B_c([0])$, $\Delta_c([0] \rightarrow [1])$ and $\Delta_c([1] \rightarrow [2])$ become eligible for garbage collection. However, if Condition 3 is instead enforced with respect to physical provenance, these blocks must be retained because they are part of the physical provenance of blocks $B_c([2])$ and $B_e([0,2,1])$.