Nova: Continuous Pig/Hadoop Workflows

Christopher Olston
Yahoo! Research, Sunnyvale, CA 94089, USA olston@yahoo-inc.com

Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B. N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell
Yahoo! Nova Development Team, Sunnyvale, CA 94089, USA
{gic, laukik, fcliu, yhan, mlarsson, anew, balav, vsankar, sseth, tianchao, topher}@yahoo-inc.com

Xiaodan Wang*
Johns Hopkins University, Baltimore, MD 21218, USA xwang@cs.jhu.edu

ABSTRACT

This paper describes a workflow manager developed and deployed at Yahoo called *Nova*, which pushes continually-arriving data through graphs of Pig programs executing on Hadoop clusters. (Pig is a structured dataflow language and runtime for the Hadoop map-reduce system.)

Nova is like data stream managers in its support for stateful incremental processing, but unlike them in that it deals with data in large batches using disk-based processing. Batched incremental processing is a good fit for a large fraction of Yahoo's data processing use-cases, which deal with continually-arriving data and benefit from incremental algorithms, but do not require ultra-low-latency processing.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Internet companies such as Yahoo, as well as many other kinds of organizations, continuously process large incoming data feeds to derive value from them. Examples at Yahoo include:

- Ingesting and analyzing user behavior logs (e.g. clicks, searches), to refine matching and ranking algorithms for search, content and advertising. Many steps are involved, including session inference, named entity recognition, and topic classification.
- Building and updating a search index from a stream of crawled web pages. Some of the numerous steps are de-duplication, link analysis for spam and quality classification, joining with click-based popularity measurements, and document inversion.
- Processing semi-structured data feeds, e.g. news and (micro-)blogs. Steps include de-duplication, geographic location resolution, and named entity recognition.

Processing along these lines is increasingly carried out on a new generation of flexible and scalable data management platforms, such as Pig/Hadoop [1, 4]. Hadoop is a scalable, fault-tolerant system for running individual map-reduce [10] processing operations over unstructured data files. Pig adds higher-level, structured abstractions for data and processing. In Pig, a language called Pig Latin [21] is used to describe arbitrary acyclic data flow graphs comprised of two kinds of operations: (1) built-in relational-algebra-style operations (e.g. filter, join); and (2) custom user-defined operations (e.g. extract web page hyperlinks, compute quantiles of a set of numbers).

Despite the success of Pig/Hadoop, it is becoming apparent that a new, higher, layer is needed: a *workflow manager* that deals with a graph of interconnected Pig Latin programs, with data passed among them in a continuous fashion. Given that Pig itself deals with graphs of interconnected data processing steps, it is natural to ask why one would layer another graph abstraction on top of Pig. It turns out that this two-layer programming model enables key scheduling and data handling capabilities:

- **Continuous processing.** The outer workflow manager layer can simulate temporally-evolving data on top of the inner Pig/Hadoop layer, which merely deals with transforming static input data into static output data. In the simple case, input data is replaced in a wholesale fashion, and processing is restarted from scratch. In advanced cases, workflow components are structured as incremental view-maintenance algorithms [14], with the workflow manager keeping track of "delta" data sets along various input, intermediate and output points, routing them to the workflow components in the right order and in ap-

*Work done during a summer internship at Yahoo!

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

propriate batches, and folding them into base data sets as needed.

- **Independent scheduling.** Different portions of a workflow may be scheduled at different times/rates. For example, global link analysis algorithms may only be run occasionally due to their costly nature and consumers' tolerance for staleness. On the other hand, the pathway that ingests new news articles, tags them with (somewhat stale) link analysis scores, and folds them into an index for serving, needs to operate (almost) continuously.

Other important capabilities that are a good fit for the workflow layer include:

- **Cross-module optimization.** An overarching workflow manager can identify and exploit certain optimization opportunities. For example, given two workflow components that consume a common input and wind up being scheduled around the same time, it can be beneficial to merge them dynamically to amortize the data reading cost. Other, more aggressive multi-query optimization [26] strategies can be employed, as well as automatic *pipelining*: connecting the output of one module directly to the input of a subsequent module (subject to scheduling and fault-tolerance considerations), to avoid the overhead of materializing the intermediate result in the file system[1].
- **Manageability features.** It can help human operators manage the workflow programming, execution and debugging lifecycle by keeping track of versions of workflow components [11], capturing data provenance and making it easily queriable [8], and emitting notifications of key events such as a workflow component finishing or failing.

We have built a system at Yahoo called *Nova* that provides these features, and is used in production. For practical reasons, Nova was designed as a layer on top of an unmodified Pig/Hadoop software stack. We believe Nova is a good design given this constraint. It is likely that, given the ability to modify the underlying Pig/Hadoop systems, one would find more efficient designs, e.g. more efficient ways to manage state for continuous processing [17].

1.1 Related Work

Data processing workflows have been studied extensively in the scientific workflow literature [18], including a recent project that integrates Hadoop with a scientific workflow manager [27]. In the Internet data processing space, motivated by some of the points listed above, several Hadoop-based workflow managers are starting to emerge (e.g. Cascading [22] and Oozie [3]). What sets Nova apart is its support for stateful continuous processing of evolving data sets. To avoid replicating basic workflow capabilities, Nova is layered on top of Oozie, which handles dependency-aware batch execution of sets of workflow elements, automatic retry and failure management, and many other core features.

Non-workflow-based approaches to continuous processing in a map-reduce-like environment include [15, 17, 25]. Map-reduce-style systems cannot achieve the ultra-low latencies required in some contexts; data stream management systems [12] are a better fit there. The recent MapReduce Online system [9] is an attempt to unify the stream and map-reduce paradigms.

Google's Percolator [24] performs transactional updates to BigTable [7] data sets, and uses triggers to cascade updates in a manner similar to a workflow. Like data stream systems, Percolator targets applications that require updates to be applied with very low latency, and so Percolator applies updates in an *eager* fashion (although under the covers, BigTable worker nodes buffer recent updates in memory). Nova, on the other hand, opts to accumulate many updates and apply them in a *lazy*, batched (and non-transactional) fashion with the aim of optimizing throughput at the expense of latency. Also, Nova sits on top of Pig and hence re-uses the Pig Latin data processing abstraction.

1.2 Outline

The remainder of this paper is structured as follows. We begin by describing Nova's abstract workflow model in Section 2, and then tie the model to Pig/Hadoop in Section 3. The architecture of the Nova workflow manager is described in Section 4. Some performance measurements are reported in Section 5. There are many avenues for future work, and we list some of them at the end of the paper in Section 6.

2. ABSTRACT WORKFLOW MODEL

A *workflow* is a directed graph with two kinds of vertices: *tasks* and *channels*. Tasks are processing steps. Channels are data containers. Edges connect tasks to channels and channels to tasks; no edge can go from a task to a task or from a channel to a channel.

Figure 1 shows an example workflow, with tasks depicted as rectangles and channels depicted as cylinders. This workflow identifies unique news articles in an incoming RSS feed. The "template detection" task looks at all articles from the same web site and identifies common template patterns, which are to be factored out of the de-duplication process (i.e. two news articles are considered duplicates if they have different template content but the same "main content"). "Template tagging" tags the portion of each article that is thought to be a template, according to the site templates output by "template detection." "Shingling" applies a hash digest technique used extensively for de-duplication in the web domain, called *shingling* [6], to the non-template content of each article. Finally, the "de-duping" task compares each incoming article's shingle hash against the hashes that have been seen previously; if the article's hash is new the article is sent to the output (and the hash is stored in the "shingle hashes seen" channel for future reference); otherwise the article is discarded.

The behavior of each task, in terms of how new input data is folded into the computation, is dictated by the edge annotations (ALL, NEW, B and Δ), which are called *consumption and production modes*. Informally speaking, ALL reads a complete snapshot of the data from a given input channel; NEW only reads data that is new since the last invocation; B emits a new, full snapshot to a given output channel; and Δ emits new data that augments any existing data. The workflow in Figure 1 illustrates four common patterns of processing:

- **Non-incremental** (e.g. template detection). Process input data from scratch every time.

[1]A less invasive alternative is to lower the level of redundancy for intermediate files.

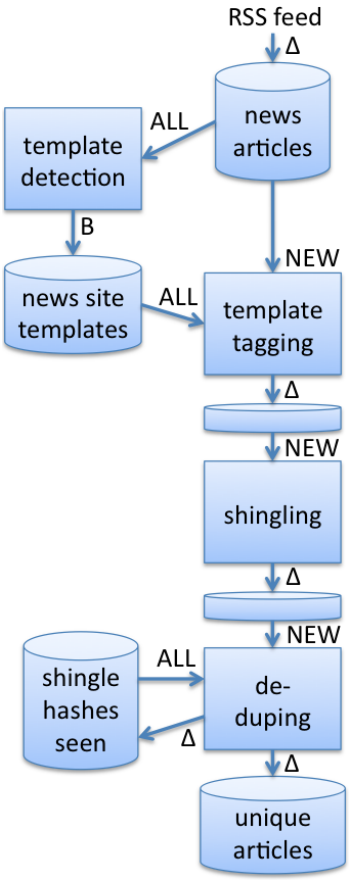

Figure 1: Example workflow.

- **Stateless incremental** (e.g. shingling). Process just the new input data; each data item is handled independently, without reference to any state.
- **Stateless incremental with lookup table** (e.g. template tagging). Process just the new input data independently of prior input data items; a side "lookup table" may be referenced.
- **Stateful incremental** (e.g. de-duping). Process just the new input data, but maintain and reference some state about prior data seen on that input.

Sections 2.1 and 2.2 are dedicated to explaining the data model and task consumption/production modes in detail. The rest of Section 2 discusses task scheduling policies, and some important space and time optimizations.

2.1 Data and Update Model

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.

Data blocks are immutable, and so as channels accumulate data blocks their space footprint can grow without bound. This situation is addressed by special compaction and garbage collection operations, described in Section 2.4.

There are two types of blocks: *base blocks* and *delta blocks* (bases and deltas, for short). A base block contains a complete snapshot of data on a channel as of some point in time. Base blocks are assigned increasing sequence numbers, and are denoted $B_1, B_2, \ldots, B_n$. Every channel is seeded with an initial, empty base block B_0.

Delta blocks are used in conjunction with incremental processing. A delta block contains instructions for transforming a base block into a new base block, e.g. by adding, updating or deleting records or columns. A delta block that transforms base B_i into base B_j (where $i < j$) is denoted $\Delta_{i \to j}$.

The process of applying a delta block to a base block, to form a new base block, is called *merging*, written $M(B_i, \Delta_{i \to j}) = B_j$. The reverse transformation, whereby two base blocks are compared to create a delta block that transforms one to the other, is called *diffing*: $D(B_i, B_j) = \Delta_{i \to j}$. Lastly, a *chain* function $C(\cdot)$ is used to combine multiple delta blocks: $C(\Delta_{i \to j}, \Delta_{j \to k}) = \Delta_{i \to k}$. A common pattern is for merging and chaining to be used together, to combine a base block with a sequence of delta blocks, as in $M(B_i, C(\Delta_{i \to j}, C(\Delta_{j \to k}, \Delta_{k \to l}))) = B_l$.

Each channel has associated merge, chain and diff functions. These functions may vary from channel to channel, depending on the type of data and data updates each channel supports. A common and simple situation is for a channel to support only appending of records, in which case the merge and chain functions are both bag union (typically achieved via simple file concatenation) and the diff function is bag difference. (Our example workflow in Figure 1 requires only append-based channels.) However Nova supports general merge, chain and diff functions, as long as they adhere to some basic rules: The chain function is required to be associative, merge and diff must be inverses, and the following relationship must hold between merge and chain: $M(M(B_i, \Delta_{i \to j}), \Delta_{j \to k})) = M(B_i, C(\Delta_{i \to j}, \Delta_{j \to k})) = B_k$.

Aside from append-only data, perhaps the most common scenario is the *upsert* model, which leverages the presence of a primary key attribute to encode updates and inserts in a uniform way. With upserts, delta blocks are comprised of records to be inserted, with each one displacing any pre-existing record with the same key. Upserts are convenient in many situations, e.g. a crawler might emit upserts consisting of (url, content) pairs, which eliminates the need for the crawler to remember whether a particular URL is being visited for the first time or revisited, because in the latter case the new content snapshot automatically supersedes the old one in the output channel. The upsert merge and chain functions perform co-group [21] on the key attribute, and retain only the most recent record with a given key. The diff function performs a set difference.

2.2 Task/Data Interface

A task must declare, for each incoming edge from a data channel, its *consumption mode*: one of ALL or NEW[2]. Con-

[2]To support incremental join algorithms, Nova also offers a special OLD mode, which can only be used in conjunction with a NEW-mode connection to the same input channel. OLD yields a complete snapshot as of the point reached by the NEW-mode connection in the previous invocation.

sumption mode ALL denotes that each time the task is executed, it is fed a complete snapshot of the data residing on the input channel in the form of a base block. If the channel contains a base block followed by one or more deltas, the latest base snapshot is created automatically on the fly via application of the merge and chain functions. This process is transparent to the task. In our example workflow in Figure 1, whenever the "template detection" task is invoked the system merges all accumulated deltas from the RSS feed to form a single, latest, base block to feed into template detection.

Consumption mode NEW denotes that each task execution is to be fed just new data that has not been seen in prior invocations, in the form of a delta block $\Delta_{i \to j}$, where i is the highest sequence number read in the most recent successful execution and j is the highest sequence number available on the channel. If it does not exist explicitly, the block $\Delta_{i \to j}$ can be created on the fly using the diff or chain function, as needed. For each task input that uses consumption mode NEW, Nova maintains an input position cursor, in the form of the highest read sequence number. (To handle cases in which $i = j$, Nova keeps a special zero-byte file to feed to tasks as an empty delta block.[3]) In our example workflow (Figure 1), if "shingling" is invoked less often than "template tagging," then multiple template tagging output delta blocks will be chained into a single delta block to feed into shingling.

Tasks must also declare a *production mode* for each outgoing edge to a channel—either B or Δ—to indicate the type of block the task emits to that channel. In our example, each invocation of "template detection" emits a new base block that replaces any prior data in the "news site templates" channel. In contrast, each invocation of "de-duping" merely emits a delta block that augments prior data in the "unique articles" output channel.

2.3 Workflow Programming and Scheduling

Workflows are programmed bottom-up, starting with individual task definitions, and then composing them into workflow fragments called *workflowettes*. Workflowettes are abstract processing components that are not attached to specific named data channels—instead they have *ports* to which input and output channels may be connected. Channels are defined via a separate registration process from workflowettes. Once workflowettes and channels have been registered, a third process, called *binding*, attaches channels to the input and output ports of a workflowette, resulting in a *bound workflowette*.

The last step is for the user to communicate scheduling requirements, by associating one or more *triggers* with a bound workflowette. There are three basic types of triggers:

- **Data-based triggers.** Execute whenever new data arrives on a particular channel (typically a channel bound to one of the workflowette's input ports).
- **Time-based triggers.** Execute periodically, every t time units.
- **Cascade triggers.** Execute whenever the execution of another bound workflowette reaches a certain status (e.g. launched, completed successfully, failed).

[3]It is not always valid to cancel execution of a task when there is no new data on one of its inputs, e.g. consider a task that performs set difference.

Once triggered, execution of a bound workflowette is atomic, i.e. any results of executions that fail mid-way are erased and never become visible to users or other workflowettes.

Our example news de-duplication workflow (Figure 1) might be implemented as two bound workflowettes: (1) template detection; (2) template tagging, shingling and de-duping. A weekly time-based trigger might suffice for refreshing the site templates (first workflowette), whereas the tagging-shingling-de-duping pipeline (second workflowette) would likely use data-based triggers so that new news article batches are processed quickly.

Nova also permits *compound triggers*, which combine two or more triggers of the above types. A compound trigger fires once all of its constituent triggers have fired. And of course, a user can always manually request to execute a particular bound workflowette.

Notice that the notion of a "full workflow" is not explicitly captured here. In Nova, a workflow is a behavior produced by a collection of bound workflowettes that exchange data via shared channels and coordinate their execution via data-based or cascading triggers.

2.4 Data Compaction and Garbage Collection

Nova performs an important data representation optimization called *compaction*, which memoizes the result of a merge (and chain) operation. For example, if a channel has blocks $B_0, \Delta_{0 \to 1}, \Delta_{1 \to 2}, \Delta_{2 \to 3}$, the compaction operation computes and adds B_3 to the channel.

Another operation, *garbage collection*, removes unneeded blocks. In our example, after compaction is used to add B_3 to the channel, the old blocks $B_0, \Delta_{0 \to 1}, \Delta_{1 \to 2}$, and $\Delta_{2 \to 3}$ may become eligible for garbage collection. Of course, garbage collection is constrained by the cursors of tasks that consume from the channel in NEW mode (see Section 2.2). For example if a consumer has a cursor at sequence number 2 then only $B_0, \Delta_{0 \to 1}$, and $\Delta_{1 \to 2}$ can be garbage-collected; $\Delta_{2 \to 3}$ must be retained until the cursor advances. Nova also supports provenance querying (not discussed in this paper), which places additional constraints on garbage collection.

Compaction, coupled with garbage collection, has two potential benefits: (1) if delta blocks contain updates and/or deletions, then the compacted data may take up less space than the non-compacted representation; (2) ALL-mode consumers do not have to merge (as many) delta blocks on the fly. In the current Nova implementation, compaction and garbage collection are triggered manually. We are working on automated techniques to determine the best time to perform these operations, in view of optimizing certain space and/or time costs.

3. TYING THE MODEL TO PIG/HADOOP

As mentioned earlier, Nova implements the data and computation model described in Section 2 on top of Pig/Hadoop.

The content of each data block resides in an HDFS[4] file (or perhaps a directory of "part" files, which is the unit of data output by a Hadoop map-reduce job). Nova maintains the mapping from data blocks to HDFS files/directories in its metadata (see Section 4). HDFS file names are hidden from users, and Nova generates unique file names by incrementing a global counter, e.g. `/nova/block_0`, `/nova/block_1`, etc. The notion of a channel exists solely in Nova's metadata.

[4]HDFS is the Hadoop filesystem.

Each task in a workflowette is specified by a Pig Latin program with open parameters for its input and output data sets, denoted by strings beginning with `$`. For example, the Pig Latin code for the "template tagging" task in our news de-duplciation workflow (Figure 1) might look like this:

```
register news_processing_udfs.jar;

articles = $RAW_ARTICLES;
templates = $TEMPLATES;
joined = join articles by site, templates by site;
tagged = foreach joined generate TagTemplates(*);
store tagged into $TAGGED_ARTICLES;
```

where `TagTemplates()` is a user-defined function whose code resides in the JAR file imported in the first line.

Each time Nova invokes a task it binds to the task input and output parameters dynamically-constructed Pig Latin expressions. In our example, suppose the latest base block for "news site templates" is stored in HDFS location `/nova/block_25`, and there are two delta blocks in the append-based "news articles" channel that have not yet been sent through the template tagging task, stored at `/nova/block_31` and `/nova/block_32`. If execution of the tagging task is triggered, its parameters will be bound as follows:

```
$RAW_ARTICLES = union (load '/nova/block_31'),
                      (load '/nova/block_32');
$TEMPLATES = load '/nova/block_25';
$TAGGED_ARTICLES = '/nova/block_33',
```

where `/nova/block_33` is a placeholder for the output block that the task execution will produce.

As we saw in the above example, for append-based channels Nova implements delta block chaining via Pig Latin's `union` operator. In general, Nova supplies a set of templates for specifying each channel's merge, chain and diff functions. Currently Nova supports two templates for merge and chain functions: (1) `union` all n input blocks; (2) `cogroup` the n input blocks by a given key attribute k, and then apply a given user-defined function f that performs a per-key merge operation (k and f are parameters to the template). For upserts, k is the channel's primary key attribute and f chooses the record residing in the right-most non-empty bag from among the n bags created by `cogroup`. The diff templates for append and upsert follow a similar pattern.

As mentioned in Section 1.1, Nova relies on a system called *Oozie* [3] to execute DAGs of Pig Latin scripts, including some important details such as sandboxing the Pig client code, automatically re-trying failed scripts, and capturing and reporting status and error messages. Nova executes a bound workflowette by first associating Pig Latin expressions with each input and output parameter of each of the workflowette's tasks (as described above), and then sending the resulting DAG of parameter-free Pig Latin scripts to Oozie for execution and monitoring. Oozie reports the final status (success or error) back to Nova. If the bound workflowette execution results in an error, Nova erases any output blocks generated during its execution, to achieve atomicity.

3.1 File Formats and Schemas

An important detail we have glossed over is how file formats and schemas are handled. Pig does not have a system catalog, and it expects[5] file formats and schemas to be specified in `load` statements, either via an explicit in-line schema description or via a special "load function" that reads some catalog or self-describing data file and passes the schema information to Pig. Zebra [5] is a self-describing format that comes with such a Pig load function.

Nova supports both manual and Zebra-based file format and schema specification. In the manual case, the user must annotate each task's output parameter (e.g. `$TAGGED_ARTICLES`) with a file format and schema. Nova keeps track of each block's file format and schema in its metadata, and passes this information to downstream Pig tasks in the generated `load` expressions.

In principle, this mechanism facilitates schema (and file format) evolution, whereby a new version of a task can emit a different schema than an old version, resulting in blocks with different schemas on the same output channel. A downstream task would be fed the correct schema for each block. Nova also allows the merge, chain and diff functions to be specified at a per-block granularity (versus per-channel), so that task upgrades have the opportunity to adjust these functions to accommodate the new schema and handle the boundary case of comparing blocks with different schemas.

Unfortunately, Nova does not currently have support for automatically synchronizing task upgrades. For example, suppose task X is upgraded so that new output blocks contain an extra column, and we wish to upgrade a downstream task Y so that it handles the new column. Currently there is no automated way to ensure that the switch to the new version of Y occurs when the first new X output block reaches Y. Instead, at present users must synchronize the upgrades of X and Y using an onerous manual process that disrupts task scheduling, e.g. (1) de-register all triggers for X and Y; (2) manually trigger Y to clear out any old blocks between X and Y; (3) upgrade X and Y; (4) re-register the X and Y triggers.

4. WORKFLOW MANAGER ARCHITECTURE

Figure 2 shows the basic architecture of the Nova workflow manager, which is divided into several modules. For the most part, these modules represent software layers, not independent threads. (The *trigger manager* module does, however, run in a separate thread so that it can support time-based events.)

Most of Nova's modules are part of a *Nova server instance* process. The modules in a server instance are stateless; they keep their state externally, in a *metadata database* (currently, MySQL Cluster [23]). The metadata database can be shared among multiple Nova server instances, which run concurrently with no explicit synchronization (the state in the metadata database effectively synchronizes them). Client requests are load-balanced among the server instances; any load-balancing policy can be used—currently we use a simple round-robin policy. A special *watchdog* module, managed via ZooKeeper [16] leader election, detects unresponsive server instances, kills them, starts fresh replacements, and reconfigures the load balancer's routing table as needed.

Nova supports two types of clients. Human clients have access to a command-line interface and a web interface.

[5]Pig can also be used without schemas, using positional notation to refer to fields.

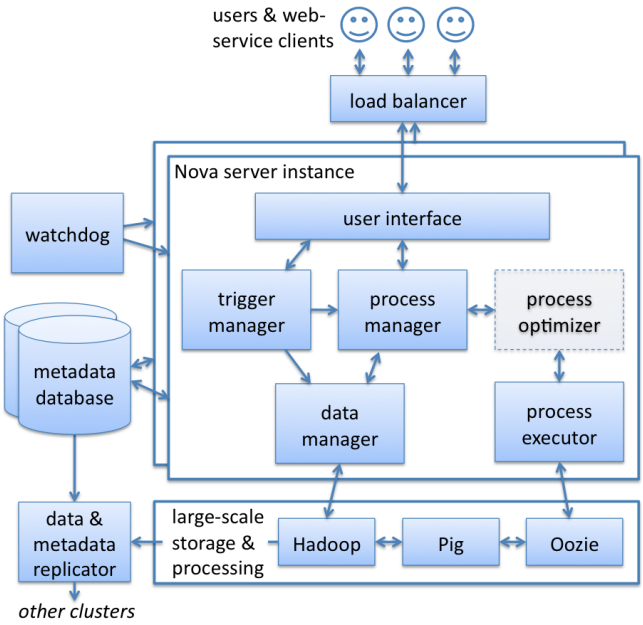

Figure 2: Nova system architecture.

Web-service clients interact with Nova via a SOAP web-services API. At Yahoo, Nova is deployed as part of a larger software environment that includes systems for *data on-boarding*, *data storage and processing* (Nova), and *data serving*. The onboarding and serving systems interact with Nova using web services.

The core Nova server modules are:

- *User interface:* This module provides API methods for registering (and deregistering) channels and workflowettes,[6] and for binding workflowettes to channels to produce bound workflowettes. Other key methods support registration/deregistration of triggers, insertion of new blocks into a channel (e.g. from the data onboarding system), monitoring workflowette execution status, and of course viewing the registered channels, workflowettes, bound workflowettes and triggers.
- *Process manager:* This module keeps track of registered workflowettes and bound workflowettes. It also responds to trigger firing events by creating an executable instance of the bound workflowette that was triggered (with the help of the data manager), and handing it off to the process executor to be run and have its execution status tracked.
- *Data manager:* The data manager maintains a list of blocks associated with each channel, as well as the mapping from blocks to underlying HDFS files/directories. It also maintains the task input cursors. When the process manager is preparing to execute a workflowette, it asks the data manager to (1) create Pig Latin expressions for loading the appropriate data from each input channel, given the current task cursors (as described in Section 3), and (2) reserve output block positions.[7] When the workflowette execution finishes, the process manager fills in or cancels the reserved output blocks (depending on whether the execution succeeded).
- *Process optimizer:* This is a placeholder for various performance optimizations on workflowette execution. One type of optimization has been implemented so far: merging pairs of workflowette executions that read the same input data and run around the same time, to amortize the data reading cost (see Section 4.2). Other types of optimizations we may consider in the future include: pipelining workflowettes that form a "chain," and adjusting the degree of parallelism to trade resource consumption against execution time.
- *Process executor:* This module forwards workflowette execution requests to Oozie (which in turn runs the constituent Pig jobs, which in turn spawns Hadoop map-reduce jobs), tracks their status, and reports the status back to the Nova process manager.
- *Trigger manager:* This module runs in its own thread, and fires triggers. As with all Nova modules, it uses the metadata database to avoid conflicts with other concurrent Nova server instances (so, e.g., if a time-based trigger fires only one instance will handle it, and others will see its status as "being handled"). Most triggers cause a workflowette to be run, via requests to the process manager. Compaction (Section 2.4) is handled by a special trigger that causes a no-op self-loop workflowette to run with consumption mode ALL and production mode B. The trigger manager also triggers garbage collection events, which are performed by the data manager.

4.1 Cross-cluster Replication

Nova includes a module, called *data & metadata replicator* in Figure 2, used to replicate Nova data and state onto other clusters, generally running in other data centers. This cross-data-center replication is asynchronous and one-way, i.e. the recipient is not an active Nova instance, but rather a (slightly lagging) "stand-by" instance. Cross-data-center replication is used for two purposes: (1) fail-over in case the primary data center becomes unavailable; (2) migration to a new data center, e.g. if the old data center is being repurposed or decommissioned.

Nova replicates its metadata database using transaction capture and asynchronous replay, similar to log-shipping. In Nova, the replication problem is complicated by the fact that persistent data is stored in two places: the metadata database and the Hadoop filesystem (HDFS), with numerous links from the metadata database to HDFS files/directories (i.e. the filesystem location of each data block). The replication mechanism carefully avoids creating dangling references at destination data-center(s), by keeping track of references

[6] Workflowettes are written in a variant of XPDL [28], which uses XML to describe a directed graph of processing steps.

[7] Currently, if two workflowettes that write blocks to the same channel run concurrently, the output blocks are sequenced in the order in which the workflowette executions began. Also, channel readers are not permitted to read "across" reserved block slots, so if writers W_1 and W_2 start in that order, and W_2 emits a block, the block is not visible to readers until W_1 finishes (either succeeds or fails). Other semantics are possible, of course; if needed we may, in the future, decide to expose control of block ordering semantics to applications.

and delaying the replay of a transaction until all referenced HDFS files have been copied.[8]

4.2 Scan Sharing

Although workflowettes are scheduled independently (via separate triggers), due to data- and time-based triggers it often happens that multiple worfkflowette executions that read the same data are triggered at (almost) the same time. Many workflowette tasks spend much of their execution time reading and writing data—often the processing itself is relatively lightweight. Hence there is an opportunity to save significant time and resources by amortizing data read costs across multiple related workflowettes.

We have implemented an experimental scan sharing capability in Nova's *process optimizer* module, which dynamically merges workflowette instances that share common inputs. It leverages Pig's ability to execute scripts containing multiple "store" operations in such a way that common operations (e.g. an initial "load") are only performed once, using branching pipelines inside the map and reduce steps [13].

Currently, workflowette merging is governed by two special annotations in the XML workflowette description: a boolean "mergeable" annotation that indicates whether it is okay to merge a given workflowette with others at execution time, and a "maximum queue time" parameter that bounds the amount of time a workflowette's execution can be delayed while waiting for other workflowettes with which to merge it.[9] Our scan sharing capability presently only supports workflowettes that use the ALL consumption mode, for which it is very easy to identify and exploit sharing opportunities; support for ones that use NEW is planned.

In our implementation, mergeable workflowette execution instances are held in a queue inside the process optimizer module for as long as their "maximum queue time" parameter allows. Whenever two workflowette instances that read from the same channel are in the queue at the same time, they are merged into a single workflowette instance using simple XML and Pig Latin rewriting. The maximum queue time of a merged instance is based on the minimum of the constituent instances. When a workflowette instance reaches its maximum queue time, it is sent to the process executor to be executed. Status messages flow back through the process optimizer module, which de-multiplexes them so that a status message about a merged instance becomes n messages, one per original instance. Hence workflowette merging is transparent to the process manager (and to the user)—in terms of semantics, but not performance, of course.

An important implication of workflowette merging is that failure of one workflowette can cause an entire combined workflowette to fail. We plan to implement a "back-off" strategy in which whenever a merged workflowette fails, each constitutent workflowette is automatically re-tried separately.

[8]Under this mechanism, destination data-centers may contain HDFS files with no corresponding metadata, but that does not affect correctness. A simple garbage collection procedure could, in principle, be used to reclaim the space they occupy, although we have not implemented such a feature because data-center transitions are rare.

[9]As future work, we would like to move to a model that is based on (soft) deadlines rather than the low-level mechanism of bounding the queue time, but a deadline-based model hinges the ability to model Pig/Hadoop execution times in a multi-tenant environment, which is elusive.

5. EXPERIMENTS

Our experiments focus on two aspects of Nova that set it apart from many other systems: incremental processing (Section 5.1) and scan sharing (Section 5.2).

All of our experiments use a real data set from one of Yahoo's document processing workflows. Records contain a significant amount of document metadata and text snippets, and hence are rather large: 1.8 KB per record, on average.

The experiments were run on a dedicated Hadoop cluster with 180 machines, although at any given time only 150–160 of the machines were functional. Each machine has an eight-core 2.50 GHz Intel Xeon processor with 16 GB of RAM, and runs Linux.

Hadoop job running times exhibit a fair amount of variance, which we attempted to mitigate by averaging across several runs.

5.1 Incremental Processing

Our incremental processing experiments are motivated by two key relational operations that are amenable to incremental evaluation: distributive aggregation, and join. Incremental distributive aggregation is accomplished in Nova via a task that emits delta blocks that contain (key, numerical increment) pairs, with the increments applied lazily (via Nova's merge feature) whenever a consumer wishes to read the full aggregate value. Incremental join also accumulates deltas that may eventually require merging, but the join task itself is more complex than the aggregation task: in addition to joining the new data from each input (left and right), it must join the new data on the left input with the old data on the right input, and vice versa.

Section 5.1.1 measures the cost to merge data blocks, and Section 5.1.2 compares the cost of an incremental join task with that of a non-incremental one.

5.1.1 Merge Overhead

To study the cost of merging data blocks, we use a no-op workflowette that simply loads and then stores a data set, and is compiled by Pig into a map-only Hadoop job (the shuffle and reduce steps are not used). As a baseline, we run the no-op workflowette over data that has been fully compacted ahead of time into a single block (i.e. no merging is necessary).

For this experiment we selected a merge function that cogroups records by a key, and then applies a UDF that examines the records associated with each key and elects to retain all of them. This merge function is like the one used for upserts (Section 3), but modified to have the property that the compacted data is the same size as the non-compacted data, thus enabling an apples-to-apples comparison against our baseline (i.e. all competitors read and write the same number of bytes).

Pig offers two alternative physical cogroup algorithms: (1) a *reduce-side* alternative that uses the Hadoop shuffle phase to group records by key; (2) a *map-side* alternative in which data blocks have been pre-organized by key[10] and can be merged in the Hadoop map phase without any shuffling.

Figure 3 shows the running time (vertical axis) as the number of blocks to be merged is varied (horizontal axis), when each block is large (ten million 1.8KB records per

[10]We use Zebra [5], a Hadoop file format that maintains data sorted and sparsely indexed by a designated key attribute.

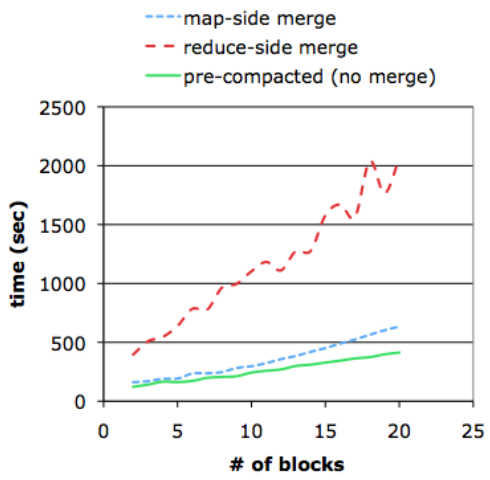

Figure 3: Running times of no-op workflowette (10M records per block).

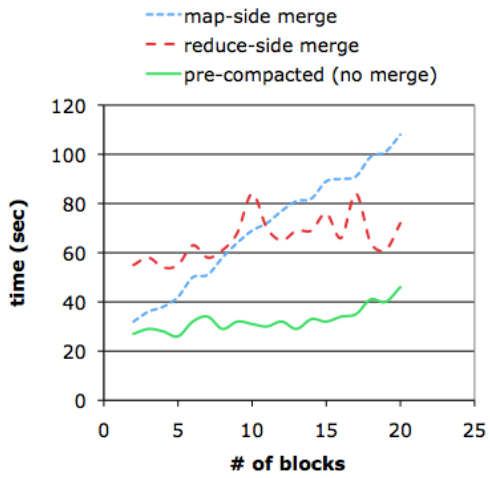

Figure 4: Running times of no-op workflowette (1K records per block).

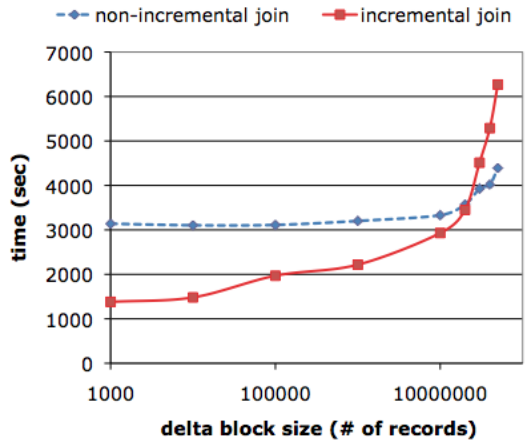

Figure 5: Join running times.

block). Map-side merging incurs a modest penalty relative to the pre-compacted baseline, as long as the number of blocks to be merged is not too high (i.e., less than ten). Reduce-side merging, on the other hand, is quite expensive due to hefty shuffle and reduce overheads.

In the case of small blocks, shown in Figure 4 (1000 records per block), the situation is quite different. Here the per-block overhead of the map-side alternative dominates, and even causes it to perform worse than the reduce-side option when the number of blocks increases beyond about ten. We have not been able to pin down the source of this per-block overhead, i.e. whether it lies mostly in opening an HDFS file or in Pig's file handling code.

An interesting question is: given a data set of a fixed size, how much difference does it make whether it is divided into a large number of small blocks, or a small number of large blocks. It turns out that for a large, 20-million-record data set the difference is minor (161 seconds for two 10-million-record blocks versus 173 seconds for 20 one-million-record blocks). However, as expected, for a small data set (20 thousand records) the difference is major: 56 seconds for two 10-thousand-record blocks versus 108 seconds for twenty one-thousand-record blocks.

These results, combined with well-known observations about the high fixed overhead of Hadoop jobs in general, imply that applications wishing to process data in small batches will likely not achieve good performance. Unlike a data stream system, Hadoop, Pig and Nova are more suitable for processing data in large batches.

5.1.2 *Incremental vs. Non-incremental Join*

Given two channels A and B with append-only data, an incremental join task computes (OLD $A \bowtie$ NEW B) $\cup$ (NEW $A \bowtie$ OLD B) $\cup$ (NEW $A \bowtie$ NEW B), which emits a delta block of the join result. This experiment compares this incremental join strategy against the non-incremental option: ALL $A \bowtie$ ALL B, which emits a base block. The physical join algorithm used in this experiment sorts the data by the join key and bulk-loads it into Zebra files (a sorted and sparsely-indexed Hadoop file format), and then runs a map-side join over the Zebra files.

The experiment assumes that each input channel contains two fully-compacted blocks: one with the old data and one with the new data. We fix the size of the old blocks at 100 million records each, and vary the size of the new (delta) blocks.

Figure 5 plots the performance of the two join strategies, with the delta block size varied on the horizontal axis using a logarithmic scale to show several orders of magnitude. As expected, when the amount of new data is small the incremental strategy is much faster (although of course it generates delta outputs rather than complete join snapshots, and obtaining a snapshot from the deltas can be expensive as we saw in Section 5.1.1). The cross-over point, at which the two strategies exhibit the same performance, occurs when the new data has about 20 million records, which is 20% of the size of the old data.

5.2 Scan Sharing

To examine the maximum potential benefits of workflowette scan sharing (Section 4.2), we studied the performance of the ideal case for scan sharing: n exact copies of a workflowette, triggered for execution at the same time. The

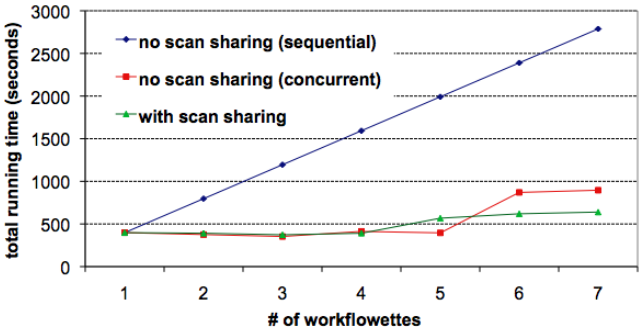

Figure 6: Running times with and without scan sharing.

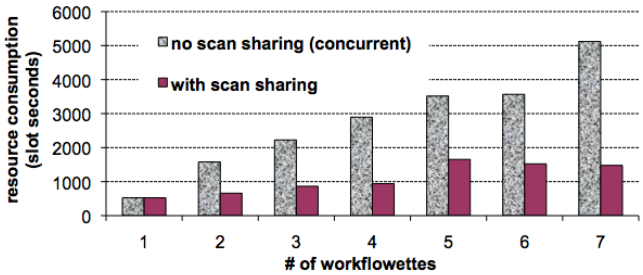

Figure 7: Resource consumption with and without scan sharing.

workflowette we used is a news de-duplication workflow used in production at Yahoo.

Figure 6 shows the total running time as the number of workflowette copies increases, under three scenarios: (1) no scan sharing, with the workflowettes run back-to-back; (2) no scan sharing, with the workflowettes submitted to Hadoop for execution all at once; (3) a single, scan-shared, execution. As the number of workflowette copies grows, the latter two scenarios both vastly outperform back-to-back independent jobs.

The key difference between the non-merged concurrent case and the merged case is their resource consumption. Hadoop manages computation resources by dividing each physical machine into a fixed number of "slots" available to run map and reduce work slices (called tasks). Hence a reasonable resource utilization metric is "slot seconds" (similar to machine seconds), i.e. the sum over all slots of the amount of time the slot is occupied by a running Hadoop task. Figure 7 shows the resource utilization (in slot seconds) of independent concurrent execution versus that of scan-shared execution. As expected, scan-shared execution has a much lower resource footprint, and hence enables greater overall system throughput.

6. SUMMARY AND FUTURE WORK

We have described a workflow manager called Nova that supports continuous, large-scale data processing on top of Pig/Hadoop. Nova is a relatively new system, and leaves open many avenues for future work such as:

- Arbitrary workflow nesting, rather than the current, rigidly tiered model.
- Better schema migration support, as motivated in Section 3.1.
- Investigating H-Base [2], or a similar BigTable [7]-inspired storage system, as the underlying storage and merging infrastructure for data channels.
- Scheduling data compaction automatically, based on some optimization formulation such as minimizing total cost while bounding wasted space.
- Automatically rewriting non-incremental workflows to execute in an incremental fashion, and perhaps even dynamically switching between incremental and non-incremental execution based on input data sizes and other factors.
- Keeping track of temporal data inconsistencies that result from delayed or unsynchronized execution of workflow components—and perhaps even incorporating consistency goals/bounds into a workflow scheduler. Preliminary work along these lines is presented in [20].
- Dealing with (minor) changes in workflow structure and task logic using incremental processing strategies, as explored in [19].

Acknowledgments

We thank the following individuals who contributed ideas, feedback, or early prototype development: Su Chan, John DeTreville, Khaled Elmeleegy, Ralf Gutsche, Christian Kunz, Patrick McCormack and Andrew Tomkins.

7. REFERENCES

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

[2] Apache. HBase: Open-source implementation of BigTable. `http://hbase.apache.org`.

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

[4] Apache. Pig: High-level dataflow system for Hadoop. `http://pig.apache.org`.

[5] Apache. Zebra: Hadoop self-describing, column-oriented file format. `http://hadoop.apache.org/pig/docs/r0.6.0/zebra_overview.html`.

[6] A. Z. Broder, S. C. Glassman, and M. S. Manasse. Syntactic clustering of the web. In *Proc. WWW*, 1997.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Computer Systems*, 26(2), 2008.

[8] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Proc. NSDI*, 2010.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[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] M. Garofalakis, J. Gehrke, and R. Rastogi, editors. *Data Stream Management*. Springer, 2009.

[13] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The Pig experience. In *Proc. VLDB*, 2009.

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

[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] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*, 2010.

[17] 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.

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

[19] C. Olston. Graceful logic evolution in web data processing workflows. Technical report, 2011. `http://i.stanford.edu/~olston/publications/workflowEvolutionTR.pdf`.

[20] C. Olston. Modeling and scheduling asynchronous incremental workflows. Technical report, 2011. `http://i.stanford.edu/~olston/publications/asynchronousWorkflowsTR.pdf`.

[21] 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.

[22] Open-Source Community. Cascading. `http://www.cascading.org/`.

[23] Open-Source Community. MySQL Cluster: A synchronously-replicated, shared-nothing database management system. `http://www.mysql.com/products/database/cluster/`.

[24] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. OSDI*, 2010.

[25] 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.

[26] T. K. Sellis. Multiple query optimization. *ACM Trans. on Database Systems*, 13(1), 1988.

[27] J. Wang, D. Crawl, and I. Altintas. Kepler+Hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. In *Proc. Workshop on Workflows in Support of Large-Scale Science*, 2009.

[28] Workflow Management Coalition. XPDL: XML process definition language. `http://www.wfmc.org/xpdl.html`.