Modeling and Scheduling Asynchronous Incremental Workflows

Christopher Olston
Yahoo! Research

ABSTRACT

We consider workflows that process very large incoming data feeds (e.g. web crawls or sky surveys) in an incremental fashion. Due to the large volume of data and relaxed application semantics, the workflow may not enforce strict temporal synchronization across the various input, intermediate and output data sets. In fact, asynchronous behavior is often programmed explicitly, to meet latency goals.

Programming of *asynchronous incremental workflows* is mostly ad-hoc, and there is a lack of tools for reasoning about and scheduling such workflows. This paper analyzes the requirements of this class of data-intensive application, and introduces formal programming and scheduling models. Our programming model gives fine-grained control over the amount of temporal asynchrony in various parts of the workflow. Our scheduler dynamically switches between synchronous and asynchronous operator variants to achieve the desired temporal semantics, and also dynamically chooses among incremental and non-incremental variants based on the amount of data to be processed and operator costs.

1. INTRODUCTION

Increasingly, organizations deploy complex workflows to transform incoming raw data feeds (e.g. web crawls, telescope images) into refined, structured data products (e.g. web entity-attribute-relationship graphs, sky object databases) [4, 20, 26, 32]. These workflows operate over vast quantities of data, which arrives in large waves and is processed *incrementally* in the manner of view-maintenance algorithms for data warehouses [18].

Typically, some portions of the workflow are trivially incremental ("stateless" processing steps like extracting noun phrases from newly-crawled pages), whereas other steps are not amenable to efficient incremental processing in the face of large input waves (e.g. computing PageRank scores [9, 14], clustering sky objects [5]) and are typically performed as periodic batch jobs. Many common operations fall in-between these two extremes: distributive/algebraic functions like counting links must reference and update a fairly large amount of state; incremental maintenance of join views requires joining newly-arrived data on each input with historical data from the other input [18].

It is generally not feasible to keep derived data sets fully synchronized with all newly-arriving input data, while still meeting the latency requirements of the application. Besides, the data entering the system is merely a stale and/or noisy reflection of the real world (asynchronously-gathered web pages and click logs with spam and robots; old photons

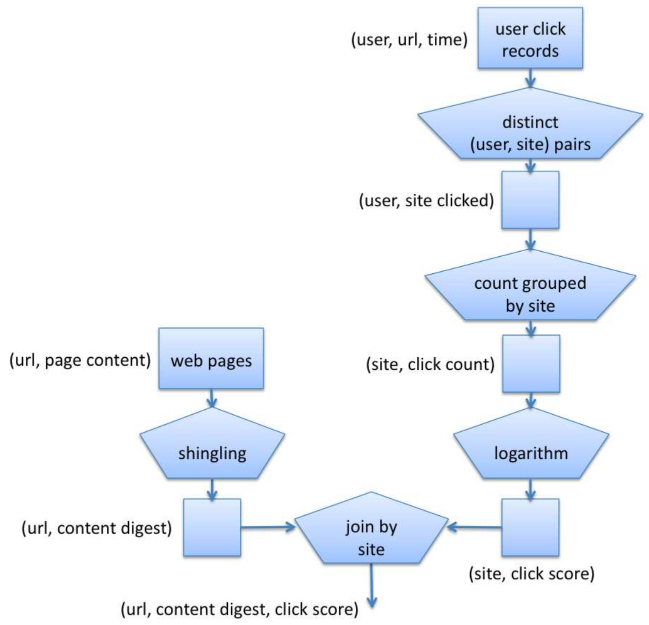

Figure 1: Web search preprocessing workflow.

with atmospheric distortion), and many of the algorithms that rely on this data and its derivations are explicitly tolerant of noise (especially machine learning algorithms and semantically loose environments like web search).

For example, a web search engine may need to expedite processing of newly-crawled web pages, even if that means labeling them with somewhat out-of-date site-level PageRank scores, and piping that temporally inconsistent joined data to further processing steps. In the astronomy domain, the latest observation of a sky object may be compared against prior observations and cluster-level statistics to flag unique temporal behaviors for follow-up observation, with the cluster statistics (and indeed the cluster model itself) updated periodically rather than continuously.

Even when there is no need to combine incoming data with out-of-date views, nonuniform latency requirements may require incoming data to be processed in non-temporal order. For example, in the web domain one might process clicks that resulted in 404 (page not found) errors ahead of earlier, successful clicks, to expedite purging of "dead" content from various caches and indices. In astronomy, one might prioritize near-earth objects, which can have a short observation window and require immediate attention. Of course, out-of-order processing can impact the semantics of downstream

time	new crawl data	new click score data	click score snapshot
Mon 1am		+(a.com, 18)	(a.com, 18)
Mon 8am	+(a.com/x, d_1)		
Tue 1am		–(a.com, 18) +(a.com, 21) +(b.com, 7)	(a.com, 21) (b.com, 7)
Tue 11am	+(a.com/y, d_2)		
Tue 5pm	+(b.com/r, d_3)		
Wed 1am		–(a.com, 21) +(a.com, 22) –(b.com, 7) +(b.com, 14)	(a.com, 22) (b.com, 14)
Wed 2pm	+(a.com/z, d_4) +(b.com/s, d_5)		

Table 1: Example crawl and click-score data, arriving as strong deltas. A "+" in front of a tuple denotes insertion; a "–" denotes deletion. Content digests are shown as $d_1, d_2, \ldots$.

time	new join output data	accumulated join output
Mon 8am + ϵ	+(a.com/x, d_1, 18)	*(any prior data ...)* (a.com/x, d_1, 18)
Tue 11am + ϵ	–(a.com/x, d_1, 18) +(a.com/x, d_1, 21) +(a.com/y, d_2, 21)	*(any prior data ...)* (a.com/x, d_1, 21) (a.com/y, d_2, 21)
Tue 5pm + ϵ	+(b.com/r, d_3, 7)	*(any prior data ...)* (a.com/x, d_1, 21) (a.com/y, d_2, 21) (b.com/r, d_3, 7)
Wed 2pm + ϵ	–(a.com/x, d_1, 21) +(a.com/x, d_1, 22) –(a.com/y, d_2, 21) +(a.com/y, d_2, 22) –(b.com/r, d_3, 7) +(b.com/r, d_3, 14) +(a.com/z, d_4, 22) +(b.com/s, d_5, 14)	*(any prior data ...)* (a.com/x, d_1, 22) (a.com/y, d_2, 22) (b.com/r, d_3, 14) (a.com/z, d_4, 22) (b.com/s, d_5, 14)

Table 2: Synchronous incremental join output.

operations, which must be taken into account.

We term workflows that introduce temporal inconsistency or out-of-orderness into derived data products as *asynchronous*. Managing such workflows poses a major challenge, as the following example demonstrates.

1.1 Example

Figure 1 depicts a simplified web search data preprocessing workflow. Data tables (including intermediate derived data products) are shown as rectangles; processing steps are shown as pentagons; arrows represent data flow. In the right-hand pathway, user click records are analyzed to find distinct user/web-site pairs (site is a simple function of URL, applied on the fly and not shown in the figure), and then determine the number of distinct users clicking on each site. A stateless "logarithm" function converts click counts into simple "click scores" used to gauge web site quality (in reality such scores incorporate other factors as well), and these scores are joined with a table of web page content digests (extracted from page content via a stateless "shingling" algorithm [6]) to associate a quality score with each

time	new join output data	accumulated join output
Mon 8am + ϵ	+(a.com/x, d_1, 18)	*(any prior data ...)* (a.com/x, d_1, 18)
Tue 11am + ϵ	+(a.com/y, d_2, 21)	*(any prior data ...)* (a.com/x, d_1, 18) (a.com/y, d_2, 21)
Tue 5pm + ϵ	+(b.com/r, d_3, 7)	*(any prior data ...)* (a.com/x, d_1, 18) (a.com/y, d_2, 21) (b.com/r, d_3, 7)
Wed 2pm + ϵ	+(a.com/z, d_4, 22) +(b.com/s, d_5, 14)	*(any prior data ...)* (a.com/x, d_1, 18) (a.com/y, d_2, 21) (b.com/r, d_3, 7) (a.com/z, d_4, 22) (b.com/s, d_5, 14)

Table 3: Asynchronous incremental join output.

page. The result is a table of URLs with content digests and click scores, which may be passed to further analysis steps or loaded into an index (a.k.a. inverted file) for searching and ranking.

Focusing on the final step (the join operation), suppose the data input to the join algorithm is as shown in Table 1. If invoked immediately after each batch of crawl data arrives, a conventional incremental join algorithm would produce the output shown in Table 2, which assigns click scores to newly-crawled URLs and also updates the click scores of URLs crawled in the past.

In the web indexing scenario, getting newly-discovered URLs into the index quickly is more important than keeping indexed click scores perfectly accurate. Hence, rather than a full incremental join of URLs with click scores, an approximate join is performed that tags new URLs with a snapshot of the click score table (perhaps indexed by site). The click score snapshot is updated only periodically (e.g. daily), and the updating must be done without disrupting any ongoing approximate join with URLs (e.g. via two concurrent versions of the click score table). The asynchronous join output is shown in Table 3. Unlike in the synchronous case (Table 2), the asynchronous output does not update the click scores of pages crawled in the past.

Unfortunately, the asynchronous join leads to unbounded staleness in click scores (a URL crawled a long time ago would be associated with a very out-of-date click score). Although a certain degree of staleness is acceptable in our application scenario, unbounded staleness is not. It is therefore necessary to occasionally (e.g. weekly) re-scan URLs that have been processed in the past and join them with newer click scores, perhaps via a full batch join to produce a replacement output table.

When bringing the click scores up to date, enqueued click records are processed through the chain of steps in the right-hand column of Figure 1 using incremental or batch processing at each step. The incremental versus batch choice depends on the amount of accumulated data, the data distributions, access methods used, and other factors affecting processing costs.

A related issue is the choice of data encoding for each intermediate data set—e.g. full-table snapshots, insertions and deletions of full records, upserts (insertion/replacement of records identified by a key), or (key, delta) representations like (site, click count increment)—which may need to change dynamically based on the processing strategies of downstream operations.

Overall, although the workflow is conceptually straightforward (Figure 1), under the covers one must rotate among several alternative data processing and representation options, while taking into account their efficiency, latency and consistency implications.

1.2 Contributions

This paper presents a formal model for representing the processing and data elements of asynchronous, incremental workflows, which may produce out-of-order or temporally incongruent data views. The paper also gives a model for scheduling such workflows, to minimize latency along critical pathways while adhering to custom temporal inconsistency bounds. Lastly, it proposes and evaluates a scheduling algorithm based on the model that automatically and dynamically chooses among operator variants and data representations.

1.3 Related Work

There are three main categories of systems that intersect with this work: data stream management, data warehousing and scientific workflow. To our knowledge, in none of these areas has the notion of temporal misalignment within derived data sets been explored. (Most systems either enforce synchronous behavior, or have "anything goes" semantics that place the onus on the programmer to reason about temporal semantics.) We elaborate below.

Data Stream Management. Data stream management systems [16] are similar in some respects to our scenario, i.e. they manage the flow of data through a network of stateful incremental operators, and indeed some aspects of this paper may be applicable to the DSMS context. However, data stream systems focus on near-real-time processing and sliding-window semantics, whereas our scenario has less stringent real-time requirements but must process data sets in their entirety (i.e. landmark windows from time zero). Moreover, to our knowledge no data stream system permits asynchrony of the form studied in this paper (e.g. non-temporally-aligned join). On the scheduling side, the Aurora QoS-driven scheduler [1, 7] is perhaps the closest to our work, but again it does not deal with asynchronous joins.

Incremental View Maintenance. Incremental view maintenance techniques [18], often studied in the data warehousing context, obviously share our focus on incrementality. The view maintenance literature considers asynchrony of view data relative to base data [13, 23, 30], but the form of asynchrony studied in this paper, in which views incorporate different base tables as of different points in time, is generally prohibited or masked from applications. Moreover, most work on view maintenance does not consider networks of cascaded views (this is the focus of scientific workflow, discussed next).

Scientific Workflow. Most scientific workflow environments have no explicit support for incremental processing (beyond trivial "stateless" processing). Exceptions include [25, 31]. To our knowledge there has been no work on supporting asynchronous processing with bounded data inconsistency in scientific workflows, other than the special case of guaranteeing zero inconsistency via strict synchronization primitives [25].

1.4 Outline

The remainder of this paper is structured as follows. Section 2 presents our formal model of asynchronous, incremental workflows. Then, Section 3 shows how to implement the standard relational operators in our model. Section 4 describes real-world-inspired workflows and their scheduling requirements. Scheduling is covered in Sections 5 (scheduling model), 6 (scheduling algorithm), and 7 (evaluation).

2. MODEL

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

- follow the "synchronous data-flow" (SDF) model of computation [24],
- are deployed for an extended period of time in a production environment, and
- incorporate newly-arriving input data in large batches using incremental algorithms.

Examples include continuously-running SDF scientific workflows [24], incremental ETL processes [22], web information extraction workflows [8], and continuous bulk processing models for map-reduce-like environments [2, 19, 28].

The specific workflow model we use is as follows. A *workflow* is a directed acyclic graph whose vertices are *data channels* and *operators*, as shown in Figure 3 (in all figures in this paper, data channels are depicted as rectangles, and operators as ovals; pentagons represent a conglomeration of operators, explained later in Section 3). A data channel is a pathway along which data flows between operators, from an external data source to an operator, or from an operator to an external data consumer. Data on a channel may be stored, pipelined, or perhaps not created at all, depending on the scheduling and execution parameters of the workflow (discussed later). Operators connect to data channels, and data channels connect to operators (or external sources/consumers); no other connections are allowed.

Data channels whose data is supplied by external sources are called *input channels*; other channels are called *derived channels*. Derived channels whose data is read by external consumers are called *output channels*; others are *intermediate channels*. A workflow may have multiple paths between a pair of data channels, representing alternative but semantically equivalent data processing pathways that present an opportunity for dynamic optimization.

2.1 Time and Data Consistency

As in temporal databases [27] and data stream management systems [3], each data channel contains a representation of a time-varying relation[1] R, i.e. a mapping from time to a relation snapshot according to a global clock.[2] In par-

[1]For simplicity of exposition we assume that data conforms to the relational model (with bag semantics), although this paper does not rely heavily on this assumption. Also, tuples in a relation may contain application-assigned timestamps (e.g. the time at which a particular web page was downloaded), which are orthogonal to system-managed time and are only assigned or interpreted at the application level.

[2]It may be a physical clock or a logical clock, i.e. a counter.

temporal provenance	$T^{\vdash}$	$T^{\dashv}$	consistent?
$T_1 = \langle\{\text{Mon 8am}\}, \{\text{Mon 1am}\}\rangle$	Mon 8am	Tue 1am	yes
$T_2 = \langle\{\text{Mon 8am}\}, \{\text{Tue 1am}\}\rangle$	Tue 1am	Tue 11am	yes
$T_3 = \langle\{\text{Tue 11am}\}, \{\text{Mon 1am}\}\rangle$	Tue 11am	Tue 1am	no
$T_4 = \langle\{\text{Wed 2pm}\}, \{\text{Mon 1am, Tue 1am, Wed 1am}\}\rangle$	Wed 2pm	Tue 1am	no

Table 4: Temporal provenance of possible crawl-clickscore join output snapshots.

ticular, a relation snapshot has an interval of validity $[t_1, t_2)$, where t_1 and t_2 are timestamps from the global clock.

An input relation being updated over time has a series of abutting intervals of validity: $[t_1, t_2), [t_2, t_3), [t_3, t_4), \ldots$. Since the right endpoint of each interval is determined by the arrival time of the next update, a relation snapshot is identified by the left endpoint, e.g. t_1. For a given snapshot timestamp t, $t^{\dashv}$ denotes the right end-point of its validity interval, e.g. $t_1^{\dashv} = t_2$.[3] The right endpoint of the most recent snapshot's interval of validity has yet to be determined, and is denoted by the special marker *now*. In comparisons and computations, *now* evaluates to the current clock time, which has the property that for any update timestamp t, $t < now$.

A snapshot of an intermediate or output relation is defined by its *temporal provenance*, i.e. the timestamps of the input relation snapshots from which it was derived. (For convenience, we sometimes shorten "temporal provenance" to "provenance," and refer to a snapshot by its provenance metadata.) Consider the n paths from some input channel to a given derived channel C. Let the input channel at the source of the ith such path be denoted $I(C, i)$. The temporal provenance of a relation snapshot S on C is denoted $T = \langle T[1], T[2], \ldots, T[n]\rangle$, where $T[i]$ is the set of timestamps of $I(C, i)$ snapshots that are reflected in S.

The temporal provenance of the click score snapshot shown in the top-right cell of Table 1 is $T = \langle\{\text{Mon 1am}\}\rangle$. For the synchronous join output snapshot in the bottom-right cell of Table 2, $T = \langle\{\text{Wed 2pm}\}, \{\text{Wed 1am}\}\rangle$. For the bottom-right cell of Table 3 (asynchronous join output), $T = \langle\{\text{Wed 2pm}\}, \{\text{Mon 1am, Tue 1am, Wed 1am}\}\rangle$, because that snapshot reflects the latest crawl data combined with click score data drawn from three different snapshots (Mon 1am, Tue 1am and Wed 1am)—e.g. three different click score values for a.com are present (18, 21 and 22).

Consider a snapshot S with temporal provenance T. Let $T[i]^{\vdash} = \max_{t \in T[i]} t$, $T[i]^{\dashv} = \min_{t \in T[i]} t^{\dashv}$, $T^{\vdash} = \max_{1 \le i \le n} T[i]^{\vdash}$ and $T^{\dashv} = \min_{1 \le i \le n} T[i]^{\dashv}$. The intersection of the validity intervals of all input snapshots from which S derives is $[T^{\vdash}, T^{\dashv})$. S is *consistent* iff $T^{\vdash} < T^{\dashv}$. The condition $T^{\vdash} < T^{\dashv}$ implies the existence of at least one point in time $t \in [T^{\vdash}, T^{\dashv})$, such that S contains the latest data from all relevant workflow inputs as of time t.

Table 4 shows the temporal provenance of some possible crawl-clickscore join snapshots, gives their $T^{\vdash}$ and $T^{\dashv}$ values, and states whether they are consistent. Intuitively speaking, a snapshot with provenance T_3 is inconsistent because it combines crawl data from validity interval [Tue 11am, Tue 5pm) with click score data from a disjoint validity interval [Mon 1am, Tue 1am). A snapshot with provenance T_4 is inconsistent because it combines click score data from three disjoint validity intervals: [Mon 1am, Tue 1am), [Tue 1am, Wed 1am) and [Wed 1am, *now*).

[3]Note that the $^{\dashv}$ transformation is channel-specific. Channel identifiers are omitted from the notation to keep it simple; it should be clear from the context which channel is referenced.

2.2 Data Blocks

In a real implementation, data associated with a relation R is manifest as a set of *data blocks* that accumulate on R's data channel over time. A data block contains one or more tuples and is the atomic unit of data from the perspective of scheduling workflow operations and reasoning about temporal (in)consistency. Data blocks may vary widely in size, from a few bytes to hundreds of terabytes, as we shall see (a data block in our model is not a unit of physical storage, i.e. it is not the same as a disk block or disk page). They come in three varieties:

- **Base data block:** $B(T)$**.** The tuples comprising the full content of the relation with temporal provenance T.
- **Strong delta data block:** $\Delta(T_1 \triangleright T_2)$**.** A set of records to be added, and a set of records to be deleted, to convert a base $B(T_1)$ to a later base $B(T_2)$, where $T_1 \prec T_2$.[4] Delta block $\Delta(T_1 \triangleright T_2)$ is said to *be relative to* base $B(T_1)$. A strong delta that contains no deletions is called *monotone*, and is denoted $\Delta^+(T_1 \triangleright T_2)$.
- **Weak delta data block:** $\delta(T_1 \triangleright T_2)$**.** Any piece of data such that one can generate $B(T_2)$ from $B(T_1)$ and $\delta(T_1 \triangleright T_2)$.

Lossless compression is permitted for any type of block. For example with strong deltas an in-place record update is semantically represented as a deletion/insertion pair, but may be physically encoded such that unchanged fields are only stored once. Weak deltas permit much more compact representations, but limit opportunities for downstream operators to deal solely with the delta representation (without reading bases). Whereas the base and strong delta representations are built into the system, weak delta representations are added as custom extensions (in the presence of multiple weak delta representations we differentiate them using subscripts: $\delta_1, \delta_2, \ldots$).

A simple example of a weak delta representation is *upserts*: a set of (key, new value) pairs such that if an old record with a given key is present in the base, the new value supersedes the old value. Upserts are quite common in practice, e.g. (URL, current page content) pairs emitted by a web crawler such that a new snapshot of a page replaces any prior snapshot. Another common weak delta representation arises in the presence of incremental aggregation operators, which may emit (key, increment) pairs, e.g. {(`cnn.com`, +5), (`yahoo.com`, -2)} applied to a base {(`cnn.com`, 28), (`yahoo.com`, 43), (`harvard.edu`, 36)}.

[4]$T_1 \prec T_2$ iff for all $1 \le i \le n$, $T_1[i]^{\vdash} \le T_2[i]^{\vdash}$ and for some $1 \le j \le n$, $T_1[j]^{\vdash} < T_2[j]^{\vdash}$, i.e. T_2 is not "behind" in time relative to T_1, and T_2 is temporally "ahead" of T_1 with respect to at least one workflow input.

2.3 Operators

Recall that in a workflow, data channels are connected by operators, which process data. Upon being invoked, an operator consumes zero or more data blocks from each of its inputs, and emits zero or more data blocks to each of its outputs. Each operator has an *input signature*, which constrains the types (among $\{B, \Delta, \Delta^+, \delta_i\}$) and consistency properties of blocks it is willing to consume. An operator's *output signature* gives the output block types and instructions for assigning temporal provenance to output blocks.

For example, consider a non-incremental (batch) join operator that reads mutually consistent base blocks from a pair of input channels C_1 and C_2, and writes a base block containing the result of the join into a third channel C_3. The input signature is: $B(T_1); B(T_2)$[5] with a consistency constraint on $T_1 \oplus T_2$ (the symbol $\oplus$ denotes list concatenation). The output signature is: $B(T_1 \oplus T_2)$.

An incremental join operator has input signature: $B(T_1), \Delta(T_1 \rhd T_2); B(T_3), \Delta(T_3 \rhd T_4)$[6] with consistency constraints on $T_1 \oplus T_3$ and $T_2 \oplus T_4$. Its output signature is: $\Delta((T_1 \oplus T_3) \rhd (T_2 \oplus T_4))$. If invoked on the example data in Table 1 following every injection of crawled data, this operator produces the output data shown in the second column of Table 2.

In Section 1.1 we alluded to an asynchronous join operator that combines delta blocks from the left input (e.g. the latest crawled pages) with base blocks from the right input (e.g. a click score relation snapshot). This operator has input signature $\Delta(T_1 \rhd T_2); B(T_3)$ (with no consistency constraints) and output signature $\Delta((T_1 \oplus T_3) \rhd (T_2 \oplus T_3))$. If invoked on the example data in Table 1 following every injection of crawled data, it produces the output data blocks in the second column of Table 3. Assuming the crawl data arrival prior to Mon 8am occurred at Sun 3pm, the temporal provenance of these blocks are[7]:

- ⟨Sun 3pm, Mon 1am⟩ $\rhd$ ⟨Mon 8am, Mon 1am⟩
- ⟨Mon 8am, Tues 1am⟩ $\rhd$ ⟨Tue 11am, Tue 1am⟩
- ⟨Tue 11am, Tues 1am⟩ $\rhd$ ⟨Tue 5pm, Tue 1am⟩
- ⟨Tue 5pm, Wed 1am⟩ $\rhd$ ⟨Wed 2pm, Wed 1am⟩

2.4 Block Type Conversion

Often, data is generated in one form (e.g. strong deltas) but a subsequent operator or external consumer wishes to consume it in another form (e.g. bases). Data may be converted among the various block types as shown in Figure 2. In five of the six *data conversion* operations the old base ($B(T_1)$) is part of the input; only conversion of a strong delta to a weak delta is always doable without reference to the old base. A *conversion operator* (*converter* for short) that transforms a data block in representation X to a data block in representation Y is denoted c^{XY}, for example $c^{B\Delta}$ is the converter from base to strong delta. (For simplicity, the fact that a converter may read the old base is left out of the superscript notation.)

To accommodate inconsistent data, we relax the input

[5]In our notation, blocks read from different input channels are separated by semicolons.

[6]When an operator's input signature contains $\Delta(T_a \rhd T_b)$, the operator may consume any chain of deltas $\Delta(T_a \rhd T_1), \Delta(T_1 \rhd T_2), \ldots, \Delta(T_{n-1} \rhd T_n), \Delta(T_n \rhd T_b)$ (the same goes for δ).

[7]We omit the {} symbols around singleton sets.

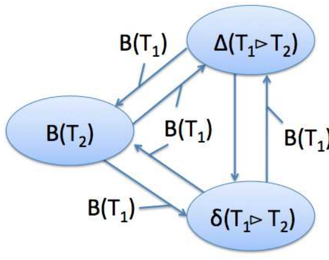

Figure 2: Block type conversion (consistent case).

signature of $c^{\Delta B}$ (and analogously $c^{\delta B}$) to: $B(T_1), \Delta(T_2 \rhd T_3)$ with the constraint that for each i either $T_1[i] = T_2[i]$ or $T_2[i] = T_3[i]$. The converter output has temporal provenance T_4 where $T_4[i] = (T_1[i] \setminus T_2[i]) \cup T_3[i]$.

Continuing our example from Section 2.3, if we start with a consistent join output base block ⟨Sun 3pm, Mon 1am⟩ and apply the four delta blocks produced by our asynchronous join (second column of Table 3), we obtain the base blocks shown in the third column of Table 3, which have the following temporal provenance:

- ⟨Mon 8am, Mon 1am⟩
- ⟨Tue 11am, {Mon 1am, Tue 1am}⟩
- ⟨Tue 5pm, {Mon 1am, Tue 1am}⟩
- ⟨Wed 2pm, {Mon 1am, Tue 1am, Wed 1am}⟩

The first of these blocks is consistent, whereas the remaining three are inconsistent because they contain a mixture of click score data from multiple points in time.

The converters $c^{B\Delta}$ and $c^{\Delta B}$ are built into the system. Other converters, which involve user-supplied weak delta representations, are specified as part of a weak delta definition. In particular, a user defining a new weak delta representation δ_i must supply at least one of $c^{B\delta_i}$ and $c^{\Delta\delta_i}$ (typically the latter), as well as at least one of $c^{\delta_i B}$ and $c^{\delta_i \Delta}$. Any unspecified converters can be constructed automatically by composing specified ones, although with no guarantee of efficiency. If multiple weak delta representations are present, the user may supply direct converters between them.

Every data channel in a workflow comes with a set of self-loops for data conversion, which are added automatically by the system. From the standpoint of workflow scheduling and execution, conversion operators act just like any other operator. Conversion operators are exempt from the acyclic workflow rule, because they merely convert among data representations. To avoid clutter, our diagrams typically do not show (all) conversion self-loops.

Discussion. The reader may be curious whether there are real-world scenarios associated with each pairwise converter shown in Figure 2. Conversion between base and (weak or strong) delta representations has obvious uses: a non-incremental operation (e.g. compute PageRank) followed by an incremental operation (e.g. update the index for URLs whose PageRank has changed), or conversely an incremental operation followed by a non-incremental one.

An example of conversion from weak delta to strong delta is as follows: A web page fetching operator emits (URL, web page snapshot) pairs, where a repeat occurrence of a particular URL denotes a newer snapshot of the page—this is a weak delta representation. A subsequent incremental

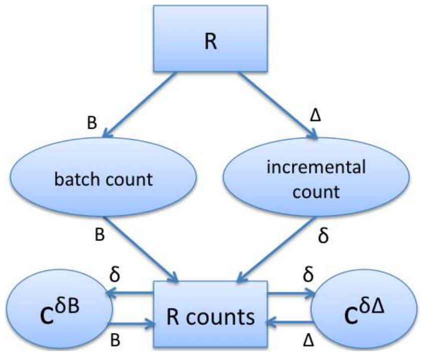

Figure 3: Distributive aggregation widget (count).

inlink counting operator needs to see additions and removals of hyperlinks as pages change over time, which requires a strong delta representation.

We have not yet encountered a compelling use case for strong delta to weak delta conversion; it is included for completeness.

3. WIDGETS AND RELATIONAL OPERATIONS

A *widget* is a sub-workflow that encapsulates multiple variants of a given operation (e.g. join), in the form of a library element that a programmer can insert into their workflow. The purpose is to permit automated, and even dynamic, selection of the operation variant based on the user's scheduling preferences, operator costs and data sizes. Even operations that have only one variant may be complex (i.e. involve multiple primitive operations and/or intermediate state), and therefore call for encapsulation into a single abstraction. Widgets may contain operators, other widgets (although recursion is not allowed), and internal data channels[8], as the following examples illustrate.

3.1 Distributive Aggregation

Figure 3 shows a widget for aggregation functions that can be computed incrementally by combining new data with the previous function output (*distributive aggregation* [17]), in this case groupwise counting. In figures in this paper, we label workflow edges with data block types (B, Δ, and so on) but to avoid clutter the consistency constraints are not shown.

The incremental pathway emits data blocks with a weak delta representation consisting of (group key, count increment) pairs (as well insertion and deletion by key, to enable creation and retirement of groups). When data follows the incremental pathway, a downstream operator will likely require that it be transformed into strong deltas or bases, which is accomplished via the converter self-loop $c^{\delta\Delta}$ or $c^{\delta B}$, respectively.

(Weak deltas will almost always be converted to strong deltas or bases so that downstream operators can make use of the data. Despite this fact, there are at least three reasons for including weak deltas as first-class components of our model: (1) conversion of weak deltas into strong deltas or bases may be batched for greater efficiency, e.g. if the producer is scheduled more often than consumers; (2) it is often more convenient to write operators that produce weak deltas (with conversion to strong deltas factored out as common code), and some legacy code may already work this way; (3) data may arrive from an external source in the form of weak deltas (e.g. web page snapshots supplied by a crawler, as described in Section 2.2).)

[8]e.g. auxiliary views for self-maintainability [29].

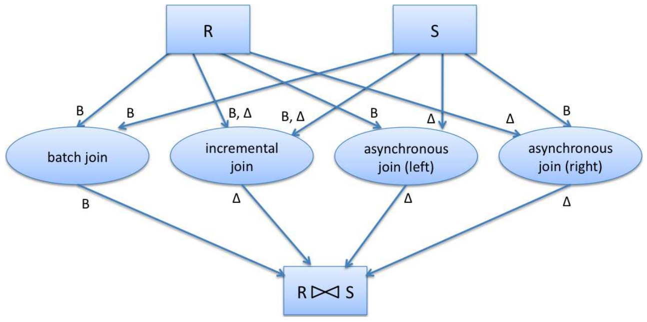

Figure 4: Join widget.

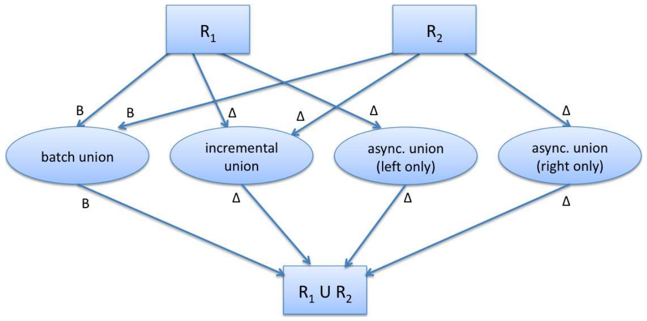

Figure 5: Bag union widget.

Widgets for *algebraic* [17] aggregation and distinct may be constructed using this distributive aggregation widget as a building block; we leave the details as an exercise for the reader.

3.2 Join

Figure 4 shows a join widget, which combines the various join operator variants described in Section 2.3:

- synchronous batch join,
- synchronous incremental join,
- two asynchronous incremental join variants: one that combines delta blocks from the left input with base blocks from the right input, and vice-versa.

3.3 Union

Figure 5 shows the bag union widget, combining:

- synchronous batch union,
- synchronous incremental union,
- asynchronous union to propagate data solely from the left (or right) input.

Unlike join, an asynchronous union operator only propagates data from one of its inputs. In the output, the temporal provenance entry for the input that was not propagated is left as an empty set. For example, consider a workflow that takes the union of two continuous crawls, where each crawl produces hourly delta blocks. Suppose we have a base block $B(\langle 9\text{am}, 9\text{am}\rangle)$ on the union output, and wish to augment it with a new delta block $\Delta(9\text{am} \triangleright 10\text{am})$ from the second crawl, without processing any data from the first

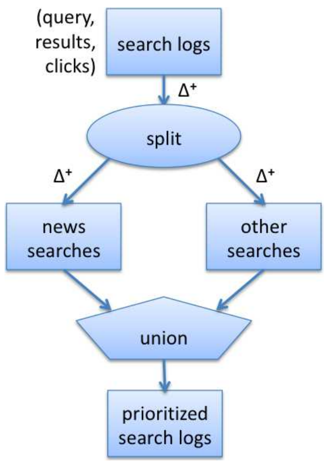

Figure 6: News relevance feedback prioritization workflow.

crawl (e.g. because data from the second crawl is more time-sensitive). Invoking the right-only asynchronous union operator, we produce $\Delta(\langle\{\},\{9\text{am}\}\rangle \rhd \langle\{\},\{10\text{am}\}\rangle)$, which, when combined with our prior output block $B(\langle 9\text{am}, 9\text{am}\rangle)$, yields the inconsistent output block $B(\langle 9\text{am}, 10\text{am}\rangle)$.

4. EXAMPLE WORKFLOWS

We present two additional asynchronous workflows from the web search domain, and discuss their scheduling requirements as a lead-in to our scheduling model (Section 5).

4.1 Feedback Prioritization

Our first example is part of a workflow that might be used for implicit relevance feedback in web search. Implicit relevance feedback is the process of improving a search ranking function based on user behavior (e.g. which results she did or did not click on) [21]. It is believed that contemporary commercial search engines employ this technique. For general web content there is no great urgency in processing relevance feedback information, but news content requires rapid adaptation. Hence a search engine might preferentially process feedback related to news, which the workflow in Figure 6 facilitates.

The input is a log of user search interactions, each containing a search query (e.g. "paris transit strike"), a list of links presented to the user in response to the query, and an indication of which of the links (if any) the user clicked on. A "split" operator divides log records into ones that contain news links among the search results, and ones that do not. The split operator is followed by a union widget (Section 3.3).

This workflow may seem rather pointless, until one considers that the temporal provenance of the output contains two entries: one for the news branch and one for the non-news branch. Hence one may request preferential processing of the news branch (details in Section 5.1).

4.2 Crawl Selection

As a final, and somewhat more elaborate, example, Figure 7 shows a simplified workflow for selecting URLs to fetch in the next iteration of an incremental web crawler [10]. Although the entire crawling system is cyclic, the web page fetching service is modeled as an external source and consumer, not a workflow operator, yielding an acyclic operator graph. The fetching service emits one (url, timestamp, error code, page content) tuple for each fetch attempt (if the attempt was unsuccessful, e.g. due to a "404: page not found" error, the page content field is set to `null`). These tuples are processed by three operators: one that logs all successful fetches, one that uses the shingling algorithm to generate a content digest of each page, and one that extracts hyperlinks that originate on the page. The extracted hyperlinks take the form of a weak delta, because the new links supersede any old links with the same source URL. A groupwise count widget determines the indegree (number of incoming hyperlinks) of each URL; indegree is a simple way to prioritize pages for crawling [12], although of course more elaborate techniques can be used.

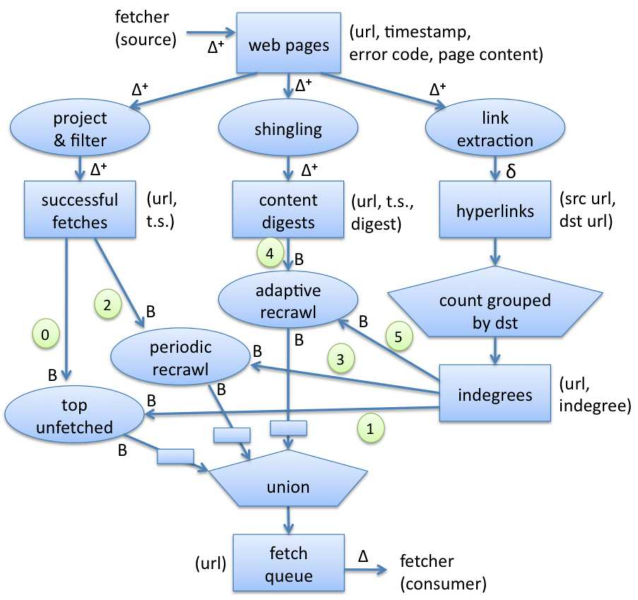

Figure 7: Web crawl selection workflow.

The workflow contains three operators that select URLs for fetching: "top unfetched," "periodic recrawl," and "adaptive recrawl." Due to the inability to assume that prior fetch requests have succeeded, as well as the need to take into account fetches requested by other operators, all three crawl selection operators are structured as periodic batch jobs (no incrementality). The "top unfetched" operator finds the top k pages by indegree that have not already been fetched. The "periodic recrawl" operator finds pages that have not been successfully fetched in the past x minutes, where x is a function of indegree. "Adaptive recrawl" finds pages that merit re-fetching based on an extrapolation of how much their content has changed based on the content evolution history, weighted by indegree, as in [11]. Note that "periodic recrawl" ensures a basic level of freshness for all crawled content, and the job of "adaptive recrawl" is to boost freshness of selected content if and when resources permit.

The three crawl selection operators emit URLs to the fetch queue, from which the fetching service reads newly-added URLs. (The three-input union widget is a simple cascade of two of the two-input union widgets shown in Figure 5.)

The scheduling requirements for crawl selection are as follows: The "top unfetched" and "periodic recrawl" crawl selection operators are to run hourly and daily, respectively, and the "successful fetches" table they read should be as up to date as possible. "Adaptive recrawl" is to run every four hours. Lastly, the indegree data referenced by all three crawl selection operators can be at most one week out of date. We introduce a formal means of communicating these requirements next.

5. SCHEDULING MODEL

A *workflow scheduler* aims to generate minimal-cost[9] operator execution plans whose output data obeys various constraints. The most basic kind of constraint is on the types of output blocks permitted. Each output channel C's external consumer specifies the set of block types that it accepts (e.g. $\{B\}$, or $\{B, \Delta\}$). The scheduler triggers operator executions such that the sequence of blocks produced on C are of the accepted type(s) and are *monotonic*. Monotonicity is defined as follows. First, let $\tau(b)$ denote the *target* temporal provenance of block b, defined as: $\tau(B(T)) = T$; $\tau(\Delta(T_1 \rhd T_2)) = T_2$; $\tau(\delta(T_1 \rhd T_2)) = T_2$. The monotonicity property requires that if a block b having $\tau(b) = T$ is emitted to channel C, then the subsequent block b' emitted to C must be one of $B(T')$, $\Delta(T \rhd T')$ or $\delta(T \rhd T')$, such that $T \prec T'$.[10] Reciprocally, the scheduler may assume that external sources inject input data blocks monotonically.

The freshness and consistency of the output data is controlled by custom constraints on the target provenance $T = \langle T[1], T[2], \ldots, T[n] \rangle$[11] of the latest data block on a given derived channel C. There are two types of target provenance constraints, one that bounds the delay with which data passes through a particular workflow pathway, and one that controls (in)consistency across pathways:

- **Freshness constraint:** $bound\_staleness(C, i, t)$
The system will attempt to maintain the invariant $T[i]^{\dashv} + t \geq now$, for some constant $t \geq 0$. This constraint bounds the degree to which provenance element $T[i]$ lags behind the latest input data on the ith workflow pathway leading to C.

- **Consistency constraint:** $bound\_inconsistency(C, t)$
The system will maintain the invariant $T^{\vdash} < T^{\dashv} + t$, for some constant $t \geq 0$. If $t = 0$, this invariant enforces strict consistency; if $t > 0$ it permits a bounded amount of inconsistency.

Consistency constraints restrict the blocks that can be fed to downstream operators or external consumers, i.e. blocks that violate one or more consistency constraints are not made available for reading (and ideally are not generated in the first place). Freshness constraints also restrict which blocks may be passed to downstream operators or consumers, but are enforced on a "best-effort" basis: the scheduler does not construct plans that violate freshness constraints, but these constraints may nonetheless become violated due to arrival of new data during plan execution (which cannot be helped). Fortunately, the goal of choosing low-processing-cost plans is aligned with the ability to repair freshness constraints quickly.

[9]We assume a standard relational-style cost model, that combines individual operator cost and selectivity estimates to produce overall plan cost estimates.

[10]As defined in Section 2.2, $T_1 \prec T_2$ iff for all $1 \leq i \leq n$, $T_1[i]^{\vdash} \leq T_2[i]^{\vdash}$ and for some $1 \leq j \leq n$, $T_1[j]^{\vdash} < T_2[j]^{\vdash}$.

[11]Recall from Section 2.1 that $T[i]$ denotes the set of timestamps of $I(C, i)$ snapshots that are reflected in the derived data, where $I(C, i)$ is the source of the ith workflow pathway leading to C.

5.1 Example Scheduling Constraints

The scheduling constraints for the search preprocessing workflow introduced in Section 1.1, expressed over the intermediate click score channel and the output channel, are:

$bound\_staleness(output, 0, 0)$
$bound\_staleness(click\ score, 0, 1\ day)$
$bound\_inconsistency(output, 1\ week)$

The scheduling constraints for our feedback prioritization workflow (Section 4.1) might be:

$bound\_staleness(output, 0, 0)$
$bound\_staleness(output, 1, 1\ day)$

Our crawl selection workflow (Section 4.2) has the most elaborate scheduling requirements. The scheduling constraints for this workflow are expressed over two intermediate channels (successful fetches and content digests), as well as the output crawl queue channel (whose temporal provenance element subscripts correspond to the pathways marked with circled numbers in Figure 7):

$bound\_staleness(successful\ fetches, 0, 0)$
$bound\_staleness(content\ digests, 0, 0)$
$bound\_staleness(output, 0, 1\ hour)$
$bound\_staleness(output, 2, 1\ day)$
$bound\_staleness(output, 4, 4\ hours)$
$bound\_inconsistency(output, 1\ week)$

6. SCHEDULING ALGORITHM

We present our scheduling algorithm in three steps:

1. A *reactive* algorithm that is invoked each time an output freshness constraint becomes violated due to arrival of new data. The algorithm identifies an operator execution plan for repairing output freshness at minimal cost, while adhering to any consistency and intermediate freshness constraints. (Section 6.1)

2. Heuristics to reduce the algorithm's search space. (Section 6.2)

3. A modification of the reactive algorithm to incorporate long-term cost considerations. (Section 6.3)

The reactive algorithm finds the optimal solution to the short-term cost minimization problem, whereas the extensions for pruning and long-term planning are heuristical.

6.1 Optimal Reactive Algorithm

The input to our scheduling algorithm consists of a workflow with selectivity and cost models for each operator, the set of data blocks materialized so far on each channel, and the scheduling constraints. The output is an execution plan consisting of a sequence of operator invocations on specific input blocks, such that all scheduling constraints are met at the completion of the plan, assuming no new data arrives in the mean time. (All intermediate results are assumed to be materialized; the pipeline vs. materialize question is left as future work.)

Our reactive scheduling algorithm is a dynamic program based on Dijkstra's shortest-path algorithm [15], with data blocks behaving like graph vertices and operator invocations behaving like graph edges.[12] For operators with more than one input, whenever the algorithm selects one input to populate, it tries all combinations of input blocks on the other inputs, drawn from the set of already-optimized blocks. The pseudocode is given in Algorithm 1.

Algorithm 1 Basic scheduling algorithm.

```
Inputs:
  workflow $W$,
  existing data blocks $B$,
  scheduling constraints $\mathcal{C}$

1: initialize priority queue $Q$ over $(block, plan, cost)$
   triples, which presents the lowest-cost entry first
2: for all $b \in B$ do
3:   insert $(b, \emptyset, 0)$ into $Q$
4: end for
5: initialize $B_{optimized} \leftarrow \emptyset$

6: while $Q$ not empty do
7:   $(b, p, c) \leftarrow Q.poll()$
8:   $G \leftarrow$ blockGen$(b, B, B_{optimized}, W, \mathcal{C})$
9:   for all $(b', p', c') \in G$ do
10:    if $Q$ contains a triple $(b', p'', c'')$ then
11:      if $c' < c''$ then
12:        $Q$.remove($(b', p'', c'')$)
13:        $Q$.insert($(b', p', c')$)
14:      end if
15:    else
16:      $Q$.insert($(b', p', c')$)
17:    end if
18:  end for
19:  $B_{optimized}$.add($(b, p, c)$)
20: end while

21: return selectOutputBlocks$(B_{optimized}, W, \mathcal{C}).plan$
```

The algorithm uses two data structures:

- Q is a priority queue of blocks that could be generated, which contains associated with each block the plan for generating the block and the estimated cost of that plan. Each entry in Q is a (block, plan, cost) triple. Lines 1–4 initialize Q and populate it with blocks that already exist, which have a cost of zero.
- $B_{optimized}$, initially empty (Line 5), will contain blocks for which the optimal plan has been identified, also in the form of (block, plan, cost) triples.

The body of the algorithm (Lines 6–20) repeatedly removes the lowest-cost entry $e = (b, p, c)$ from Q, which has the property that p is the optimal plan for obtaining block b, and enumerating ways to use p as a sub-plan in a larger plan that creates other blocks (labeled b'). Each newly-enumerated (b', p', c') triple is inserted into Q (possibly displacing previously-discovered, but more expensive, options for producing b'), and e is inserted into $B_{optimized}$. In more detail:

- Lines 7–8 remove the lowest-cost entry $e = (b, p, c)$ from Q and enumerate hypothetical blocks that can be generated by applying a workflow operator to b. The enumeration is performed by the `blockGen` subroutine, which considers all legal[13] applications of an operator $o \in W$ to b. The details of `blockGen` are straightforward, and hence omitted, except for the manner in which multi-input operators (e.g. join) are handled: when block b is considered for one of the inputs, the remaining inputs are selected from $B \cup B_{optimized}$, with all legal options exhaustively explored.[14] Each operator/input combination yields a hypothetical output block, which is checked for constraint violations; ones that satisfy all scheduling constraints in $\mathcal{C}$ are returned by `blockGen`.
- Lines 9–18 consider each enumerated block b' in turn, and check to see if Q already contains a cheaper plan for b'. If not, the new plan is inserted into Q, and any prior (more expensive) plans for b' are discarded. The result is that Q contains exactly one plan for b', namely the cheapest plan discovered so far.
- Line 19 adds entry $e = (b, p, c)$, which was removed from Q in Line 7 and is known to represent the cheapest way of generating block b, into $B_{optimized}$.

When Q is empty and there are no further plans to explore, the final step (Line 21) is to select, for each output channel, the cheapest plan for producing a block that (1) meets all scheduling constraints, and (2) is monotonic with respect to the prior block produced. By construction, all blocks in $B_{optimized}$ meet the scheduling constraints, so the `selectOutputBlocks` subroutine (not shown) simply selects, for each output channel, the cheapest plan that produces a monotonic block.

6.2 Controlling the Search Space

In practice, the above algorithm searches over a prohibitively large space of (actual and potential) data blocks. We have devised a set of heuristics for pruning the search space, to enable the algorithm to run in a reasonable amount of time:

6.2.1 Pruning Old Blocks

Our first pruning technique eliminates blocks that are older than the current processing frontier and hence are no longer useful. For a given output channel O, let the *frontier* $f(O)$ be the target of a block b whose type is accepted by all external consumers, such that all other accepted output blocks b' have $\tau(b') \preceq \tau(b)$. For a channel C that directly feeds O, $f(C)$ consists of the projection of O's output frontier to retain the portions that originate from C. The frontier is defined recursively toward the input channels in this fashion. (Channels with multiple paths to output channels have a *frontier set*; in such cases $f(C)$ is defined as the minimal frontier set element.) Channel C's *maximal pre-frontier base* $T(C)$ is a base block b with $\tau(b) \prec f(C)$ such that no other base block b' on C has $\tau(b) \prec \tau(b') \prec f(C)$. During scheduling, all blocks b'' with $\tau(b'') \prec p(C)$ are ignored.

[12]Delta chaining (Section 2.3) is modeled as a zero-cost operator.

[13]Block b must reside on the channel from which o reads, and be of the type required by o (i.e. B, Δ or δ_i).

[14]In practice the number of legal options is not large and the running time of this enumeration is reasonable, as verified in Section 7.

6.2.2 *Forming Compulsory Delta Chains*

As data arrives, long chains of successive delta blocks can accumulate on input and/or intermediate channels, which unnecessarily increase the search space for scheduling. We automatically combine, or *chain*, a sequence of abutting delta blocks $\Delta(T_a \rhd T_1), \Delta(T_1 \rhd T_2), \ldots, \Delta(T_{n-1} \rhd T_n), \Delta(T_n \rhd T_b)$ into a single block $\Delta(T_a \rhd T_b)$ for the purpose of scheduling (the same goes for δ). However, we do not form a chain that spans an intermediate temporal provenance entry T_i if either of the following conditions hold: (1) the channel contains a non-pruned base block $B(T_i)$; (2) the channel's frontier set contains T_i. In either of these cases, chaining could eliminate important processing pathways and perhaps even permanently block processing on the workflow.

6.2.3 *Propagating Constraints Upstream*

We eliminate from consideration blocks that are unlikely to lead to blocks that satisfy the downstream scheduling constraints, by propagating scheduling constraints upstream from the output channels toward the input channels, in a recursive fashion. When propagating constraints from channel C to a channel C' that feeds C, we project C's staleness bounds (if any) to retain only the provenance elements that derive from C', and simply copy C's inconsistency bound (if any). If C' has its own staleness and/or inconsistency bounds, they are combined with the ones propagated from C by taking the minimum time bound constant (t value) of each corresponding pair. Lastly, we constrain all propagated staleness bounds on channel C to be less than or equal to C's propagated inconsistency bound.

6.2.4 *Pruning Overkill Blocks*

If the scheduler enumerates a potential block b on channel C, and C already contains an actual block b' such that (1) b and b' are of the same type (e.g. both strong deltas), (2) $\tau(b') \prec \tau(b)$, and (3) b' satisfies C's propagated scheduling constraints, then b is called an *overkill* block and is pruned. Overkill blocks represent unnecessary processing work, and plans that incorporate them will ultimately not be chosen by the scheduler due to excessive cost.

6.3 Incorporating Long-term Considerations

As mentioned above, our basic scheduling algorithm is *reactive*, in the sense that it waits for a freshness constraint to be violated and then focuses on minimizing short-term costs to repair the violated constraint. Take the following example: In our crawl selection workflow we place an inconsistency bound of one week on the output channel (Section 5.1). After the initial week has elapsed, our reactive scheduling algorithm updates the "indegrees" data to be just under one week old and recomputes the output data. This approach satisfies the immediate scheduling constraints at minimal cost, but leads to updating the "indegrees" data in every iteration. A better long-term strategy is to bring the "indegrees" data fully up to date so that it can remain idle for a week.

We therefore add the following provision: ignore potential block b if another potential block b' of the same type has been enumerated (and not pruned) for the same channel, such that $\tau(b) \prec \tau(b')$. (Note that this provision does not eliminate cost-based competition among different execution strategies; there is still competition among blocks with equal or incomparable target provenance.)

7. EVALUATION

We evaluate our asynchronous processing model and scheduling algorithm with a workflow simulator. Our simulator encapsulates all aspects of the workflow model described in this paper except actual operator execution. Operator execution is modeled as a noisy process in which the running time and output data size are assigned by applying standard relational cost and selectivity models and adding significant random perturbation (a random, Gaussian-distributed increase or decrease that averages 50% of the original value).

We programmed the workflows described in Sections 1.1, 4.1 and 4.2, referred to here as `SearchPreprocessing`, `FeedbackPrioritization` and `CrawlSelection` respectively, into our simulator and assigned the scheduling constraints given in Section 5.1. Incoming data rates, operator selectivies and costs for these workflows were set based on real-world parameters.

Our simulator models an elastic computing ("cloud") environment in which computing resources are effectively unlimited but incur a linear cost, and embarrassingly-parallel set-oriented processing primitives achieve near-linear scale-up (e.g. map-reduce). In this context, both time and cost are linearly proportional to the number of CPU hours required for a given computation.

7.1 Metrics

Recall from Section 5 that while the scheduler guarantees to uphold consistency constraints at all times, freshness constraints can only be enforced on a "best-effort" basis. Hence one of the important evaluation criteria is *freshness lag*, i.e. the duration of freshness constraint violations. The other two evaluation criteria are the computation cost of the generated execution plans, and the scheduling overhead.

In detail, our evaluation metrics are:

- **Freshness lag:** the lag between the time at which a freshness constraint on an output channel becomes violated (due to arrival of new data) and the time at which it is repaired (due to completion of some processing steps). In the ideal case, the necessary computations have been performed in advance and the lag is zero. In our experiments we report the median lag in output freshness, in simulated CPU hours.
- **Computation cost:** the total cost to process data, over the lifetime of the workflow. In our experiments, this quantity is measured in simulated CPU hours.
- **Scheduling overhead:** the scheduler running time, summed across all invocations. In our experiments, the scheduler was run on a 2.53 GHz dual-core processor with 4 GB of RAM.

All measurements are averaged over several runs to reduce noise.

7.2 Results

Table 5 shows the results of running each workflow for a ten-week duration under four alternative scenarios:

- **Async/Incr:** The full processing model of this paper.
- **Sync/Incr:** Only consistent blocks are permitted.
- **Async/Batch:** Only base blocks are permitted.
- **Sync/Batch:** Consistent, base blocks only.

WORKFLOW	PROCESSING METHOD	FRESHNESS LAG (CPU hours)	PROCESSING COST (CPU hours)	SCHEDULER RUNNING TIME (cumulative seconds)
`SearchPreprocessing`	**Async/Incr**	0.0606	28,900	6.25
	Sync/Incr	27.9	214,000	2.59
	Async/Batch	615	8.26 million	0.864
	Sync/Batch	808	43.8 million	0.787
`FeedbackPrioritization`	**Async/Incr**	0.0435	6930	2.77
	Sync/Incr	2.12	7130	2.25
	Async/Batch	139	35.0 million	0.692
	Sync/Batch	193	1.34 billion	0.758
`CrawlSelection`	**Async/Incr**	81.4	364,000	10.1
	Sync/Incr	542	1.69 million	1.71
	Async/Batch	2380	81.2 million	0.743
	Sync/Batch	3890	114 million	0.667

Table 5: Workflow efficiency for asynchronous/incremental versus synchronous/batch, and dynamic programming scheduler running times, for ten-week workflow runs.

Schedule quality. In the Async/Incr case our scheduling algorithm produces the execution behaviors we hoped for, as described in Sections 1.1 and 4, which selectively avoid updating derived data sets when the existing level of temporal consistency is adequate. Consequently the median freshness lag is one to three orders of magnitude lower than in the Sync/Incr case, depending on the workflow. With the `SearchPreprocessing` and `FeedbackPrioritization` workflows, when arrival of new input data renders the current output stale, in the asynchronous case typically only a small amount of data is processed to re-establish output freshness (just new pages (not new clicks); just news-related searches), which achieves a three-orders-of-magnitude reduction in freshness lag compared to the synchronous case. With the `CrawlSelection` workflow the gain is less dramatic, but still large—in this case the decreased lag comes from only occasionally executing the expensive "periodic recrawl" and "adaptive recrawl" computations.

In terms of overall processing cost, with the `FeedbackPrioritization` workflow there is little difference between the Async/Incr and Sync/Incr cases, because all searches are processed eventually. With `SearchPreprocessing` the overall cost is much lower in the Async/Incr case relative to the Sync/Incr case, primarily due to avoiding continual symmetric incremental joins between web pages digests and click scores, in favor of one-sided incremental joins punctuated by occasional full join re-computations. Lastly, with the `CrawlSelection` workflow the reduction in total processing cost comes from running the "periodic recrawl" and "adaptive recrawl" steps fewer times.

In terms of absolute numbers, the `CrawlSelection` workflow has the highest freshness lag and processing cost, mainly because it contains non-incremental elements. Even then, forcing fully non-incremental execution makes freshness and total processing cost much worse, as shown in the Async/Batch case. In practice the `CrawlSelection` workflow is run on a large cluster of approximately 1000 machines, in which case the 81.4 CPU hours for freshness lag translates to around five minutes of processing time. The total processing cost—364,000 CPU hours over a ten-week period—translates to 13 CPU minutes per machine per hour, which gives a very comfortable operating margin.

Scheduling overhead. The right-most column of Table 5 gives total running times for our scheduling algorithm. As these numbers show, our scheduler is extremely efficient, requiring just a handful of seconds total to perform all scheduling decisions for a ten-week workflow run.

Comparison with simpler scheduling algorithms. We compare our dynamic programming scheduling algorithm against two baselines:

- GREEDY: Each time the scheduler is invoked, consider blocks that can be derived directly from existing blocks (after pruning old blocks as in Section 6.2.1 and forming compulsory delta chains as in Section 6.2.2). Select the lowest-cost block that satisfies the propagated scheduling constraints (Section 6.2.3), and return the single-operator plan that generates the selected block.
- RANDOM: Enumerate candidate blocks in the same manner as GREEDY. If any of the enumerated blocks has zero cost (e.g. a delta chain), select it; otherwise select a block at random. (The purpose of RANDOM is to establish a lower bound on scheduler performance.)

In our experience, GREEDY tends to generate many intermediate data blocks that are cheap (relative to other possible blocks), yet turn out not to be useful in producing the outputs that are required in the short-term (which hurts freshness) and/or long-term (which hurts total processing cost). The accumulation of cheap but useless blocks compounds the problem, as these blocks open up avenues to generate yet more cheap, useless blocks. This process both blows up the search space of the algorithm and leads to a spiral of worsening cost and freshness results.

Due to the search space blow-up experienced by GREEDY, we had to limit our performance comparison of different scheduling algorithms to short workflow runs. Table 6 shows a comparison of scheduling policies on twelve-hour workflow runs. Although the greedy strategy is clearly far better than random, for the reasons stated above our approach performs much better still, in terms of both freshness and total processing cost, on all workflows studied.

8. SUMMARY

This paper introduced an *asynchronous* model of incremental data processing, which arises from a real-world need for low-latency processing of critical data combined with periodic bulk processing of less critical data. Asynchronous

WORKFLOW	SCHEDULING ALGORITHM	FRESHNESS LAG (CPU hours)	PROCESSING COST (CPU hours)	SCHEDULER RUNNING TIME (cumulative seconds)
`SearchPreprocessing`	DYN. PROG.	0.0399	187	0.208
	GREEDY	17.0	255	49.8
	RANDOM	61.6	932	1.86
`FeedbackPrioritization`	DYN. PROG.	0.0485	24.5	0.183
	GREEDY	5.10	66.9	0.507
	RANDOM	19.1	413	0.374
`CrawlSelection`	DYN. PROG.	4.94	356	0.176
	GREEDY	96.2	783	18.1
	RANDOM	396	2840	0.530

Table 6: Comparison against baseline scheduling algorithms, for twelve-hour workflow runs.

data processing workflows have complex scheduling requirements and necessitate careful control over the inconsistent data they generate. Our formal model of workflow semantics and scheduling constraints simplifies the task of programming such workflows and obtaining the desired latency and consistency properties. Our scheduling algorithm produces low-latency schedules that meet the consistency requirements with very little scheduling overhead, greatly outperforming a baseline greedy approach in terms of quality of schedules produced as well as scheduler running time.

Acknowledgments

We are grateful to the Yahoo! Nova team for their inspiration and feedback, especially Laukik Chitnis, Yiping Han and Andreas Neumann. We also thank Al Olston and Anish Das Sarma for their feedback.

9. REFERENCES

[1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), 2003.

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

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[4] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *Proc. IJCAI*, 2007.

[5] C. S. Botzler, J. Snigula, R. Bender, and U. Hopp. Finding structures in photometric redshift galaxy surveys: An extended friends-of-friends algorithm. *Monthly Notices of the Royal Astronomical Society*, 349:425–439, 2004.

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

[7] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proc. VLDB*, 2003.

[8] 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.

[9] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar. Link evolution: Analysis and algorithms. *Internet Mathematics*, 1(3), 2003.

[10] J. Cho and H. García-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. VLDB*, 2000.

[11] J. Cho and H. García-Molina. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems*, 28(4), 2003.

[12] J. Cho, J. García-Molina, and L. Page. Efficient crawling through URL ordering. In *Proc. WWW*, 1998.

[13] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. ACM SIGMOD*, 1996.

[14] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Proc. WWW*, 2005.

[15] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

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

[17] J. Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

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

[19] 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.

[20] Z. Ivezic, J. Tyson, R. Allsman, J. Andrew, R. Angel, and et al. LSST: from Science Drivers to Reference Design and Anticipated Data Products. http://arxiv.org/abs/0805.2366.

[21] T. Joachims, L. Granka, and B. Pan. Accurately interpreting clickthrough data as implicit feedback. In *Proc. SIGIR*, 2005.

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

[23] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proc. ACM SIGMOD*, 1986.

[24] B. Ludascher et al. Scientific process automation and

workflow management. In *Scientific Data Management: Challenges, Technology, and Deployment*, chapter 13. Chapman & Hall/CRC, 2009.

[25] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. Towards continuous workflow enactment systems. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2008.

[26] Z. Nie, Y. Ma, S. Shi, J.-R. Wen, and W.-Y. Ma. Web object retrieval. In *Proc. WWW*, 2007.

[27] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowledge and Data Engineering*, 7(4):513–532, 1995.

[28] 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.

[29] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. PDIS*, 1996.

[30] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *Proc. ACM SIGMOD*, 2000.

[31] T. Tavares et al. An efficient and reliable scientific workflow system. In *Proc. of 7th International Symposium on Cluster Computing and the Grid (CCGrid'07)*, 2007.

[32] D. York et al. The Sloan Digital Sky Survey: Technical Summary. *Astronomy Journal*, 120:1579–1587, 2000.