# Stateful Bulk Processing for Incremental Analytics

Dionysios Logothetis
UCSD Computer Science
dlogothetis@cs.ucsd.edu

Christopher Olston
Yahoo! Research
olston@yahoo-inc.com

Benjamin Reed
Yahoo! Research
breed@yahoo-inc.com

Kevin C. Webb
UCSD Computer Science
kcwebb@cs.ucsd.edu

Ken Yocum
UCSD Computer Science
kyocum@cs.ucsd.edu

## ABSTRACT

This work addresses the need for stateful dataflow programs that can rapidly sift through huge, evolving data sets. These data-intensive applications perform complex multi-step computations over successive generations of data inflows, such as weekly web crawls, daily image/video uploads, log files, and growing social networks. While programmers may simply re-run the entire dataflow when new data arrives, this is grossly inefficient, increasing result latency and squandering hardware resources and energy. Alternatively, programmers may use prior results to incrementally incorporate the changes. However, current large-scale data processing tools, such as Map-Reduce or Dryad, limit how programmers incorporate and use state in data-parallel programs. Straightforward approaches to incorporating state can result in custom, fragile code and disappointing performance.

This work presents a generalized architecture for continuous bulk processing (CBP) that raises the level of abstraction for building incremental applications. At its core is a flexible, groupwise processing operator that takes state as an explicit input. Unifying stateful programming with a data-parallel operator affords several fundamental opportunities for minimizing the movement of data in the underlying processing system. As case studies, we show how one can use a small set of flexible dataflow primitives to perform web analytics and mine large-scale, evolving graphs in an incremental fashion. Experiments with our prototype using real-world data indicate significant data movement and running time reductions relative to current practice. For example, incrementally computing PageRank using CBP can reduce data movement by 46% and cut running time in half.

## Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems

## General Terms

Algorithms, Design, Performance


*SoCC'10*, June 10–11, 2010, Indianapolis, Indiana, USA.


## 1. INTRODUCTION

There is a growing demand for large-scale processing of unstructured data, such as text, audio, and image files. It is estimated that unstructured data is now accumulating in data centers at three times the rate of traditional transaction-based data [23]. For instance, YouTube integrates 20 hours of new video a minute, Facebook analyzes 15 TB's of information a day [20], and Internet search companies regularly crawl the Internet to maintain fresh indices. Many large-scale Internet services, such as social networking sites or cloud-computing infrastructures, analyze terabytes of system and application-level logs on a daily basis to monitor performance or user behavior [27].

These environments often require data processing systems to absorb terabytes of new information every day while running a variety of complex data analytics. This data "deluge" presents major data management challenges, and non-relational information analysis is quickly emerging as a bedrock technology for these large-scale data processing efforts. Today, parallel data processing systems, like Map-Reduce [10] and Dryad [14], offer a scalable platform that can leverage thousands of cheap PC's for large data processing tasks. For example, the social networking site Facebook uses Hive [1], a high-level relational programming language for Hadoop (an open-source Map-Reduce), to manage their 2.5 petabyte data warehouse [20].

Many of these applications must combine new data with data derived from previous batches or iterate to produce results, and state is a fundamental requirement for doing so efficiently. For example, incremental analytics re-use prior computations, allowing outputs to be updated, not recomputed, when new data arrives. Incremental/iterative variants exist both for updating the answers (views) to relational queries [3] and for non-relational algorithms such as spatio-temporal queries [19], data clustering [12, 21], and page rank [8], to name a few. Such an approach has obvious potential for large performance improvements, increasing the size of problems that may be tackled with a given hardware/energy budget.

However, current bulk-processing models limit how programmers incorporate state into their data-parallel programs, often forcing programmers to add state by hand. To avoid this complexity, they may re-use existing dataflows and re-process all data, paying for a larger compute cluster to avoid performance penalties. Alternatively, they may add state by re-reading prior outputs or storing data in an external storage service. In either case, state is outside the purview of the bulk-processing system, where it either treats it as any other input or is unaware of it, limiting the opportunities for optimization. We find (Section 5.1) that this can lead to run times proportional to total state size, not the changes to state, significantly reducing the performance gains promised by incremental algorithms. Moreover, current bulk-processing primitives are de-

signed for single-shot operation, not for environments with continuous data arrivals. Thus they provide few mechanisms for synchronizing processing across inputs or defining which data records to process next.

This paper describes the design and implementation of a system for *continuous bulk processing* (CBP). A core component of CBP is a flexible, stateful groupwise operator, *translate*, that cleanly integrates state into data-parallel processing and affords several fundamental opportunities for minimizing data movement in the underlying processing system. Additionally CBP offers powerful dataflow management primitives to accommodate continuous execution when using one or more translate operators and a scalable and fault-tolerant execution platform based on a modified Hadoop.

This paper makes the following contributions:

- **Stateful groupwise operator:** We propose a data processing operator, *translate*, that combines data-parallel processing and access to persistent state through grouping. This abstraction unifies two common programming practices: the inclusion of state to re-use prior work for incremental processing, and groupwise processing, a well-known interface for bulk-data processing.
- **Primitives for continuous bulk processing:** Continuous dataflows require precise control for determining stage execution and input data consumption. The CBP model includes primitives that support control flow and allow stages to synchronize execution with respect to multiple inputs. These features simplify the construction of incremental/iterative programs for large, evolving data sets.
- **Efficient implementation:** As we will show, emulating stateful dataflow programs with current bulk processing operators, such as those in the Map-Reduce model, leads to unacceptable performance (running time superlinear in input size). Thus, we design a custom execution system (Section 4.5) that minimizes data movement by taking advantage of constructs in the CBP model. These constructs allow the system to optimize the incremental grouping and modification of state records.
- **Applications and evaluation:** We explore CBP using a variety of processing tasks, including a simplified web crawl queue and two incremental graph processing algorithms (PageRank [8] and clustering coefficients [26]), using real-world data, including Yahoo! web and Facebook crawls. We find the CBP model can express many incremental processing constructs and that a direct implementation of the model can ensure incremental performance for a range of workloads. Our experiments that incrementally compute a web crawl queue (described in the next section) reduced the cumulative running time from 425 to 200 minutes (53%), when compared to a non-optimized approach.

## 1.1 Example

As a motivating example, consider the challenge Internet search engines face to keep indices up-to-date with the evolving web corpus. To do so, they must optimize their crawl, looking for the most valuable, rapidly changing parts of the web. This is an iterative process; they first crawl a part of the web, add it to the collection of crawled pages, rank the new and previously found links, and initiate another crawl on the highest-ranked links. A "single shot" processing approach re-processes all cached web pages after each partial crawl. While obviously inefficient (the ratio of old to new data may be ten to a thousand times), this can occur in practice due to its simplicity and the ability of bulk-processors, like Map-Reduce, to easily scale up.

Figure 1 shows a bulk-incremental workflow to compute the crawl queue for a web indexing engine. Though crawling is a complex issue, this workflow gives a high-level overview of how one might

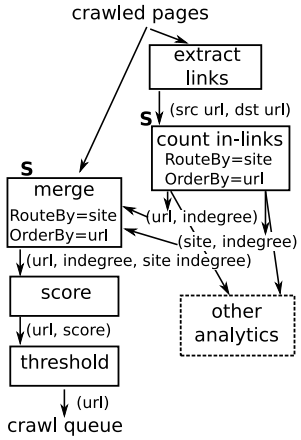



**Figure 1: A dataflow for incrementally computing a web crawl queue. Edges represent flows of data, and stages marked with an S are stateful.**

leverage state. The first processing stage, *extract links*, extracts the in-links from the raw web page text. Next, the *count in-links* stage counts the number of times particular URLs and web sites appear within the newly crawled pages. This stage has two outputs, one for each count. The *merge* stage combines those counts with the current known set of crawled pages. This stage sends new and updated URLs from the last crawl to the next two stages that score and threshold the updates. Those URLs whose scores pass the threshold are the next crawl queue.

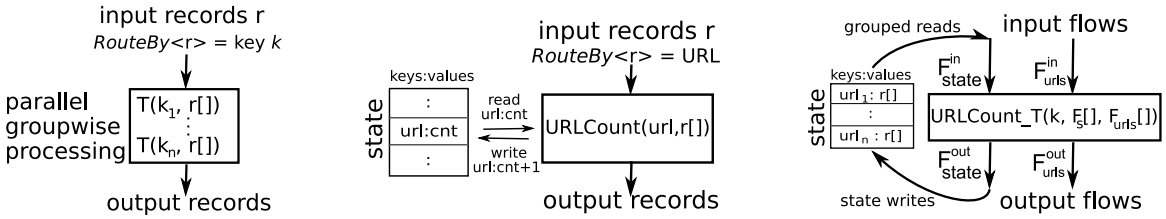
This workflow supports incremental computation in multiple ways. First, it runs *continuously*: an external crawler reads the output, crawls the new pages, and waits for the workflow to run again. As in a data stream management environment [2], edges transmit only new or updated data items, and the execution system only runs stages when there is sufficient (as defined by the programmer) data on each input edge. Second, stages provide incremental processing by leveraging persistent state to store prior or partial results (stages marked with **S**). Many analytics exhibit opportunities for this kind of incremental processing, including the standard set of aggregate operators (e.g., min, median, sum, etc.), relational operations [3], and data mining algorithms [18].

## 1.2 Related work

**Non-relational bulk processing:** This work builds upon recent non-relational bulk processing systems such as Map-Reduce [10] and Dryad [14]. Our contributions beyond those systems are two-fold: (1) a programming abstraction that makes it easy to express incremental computations over incrementally-arriving data; (2) efficient underlying mechanisms geared specifically toward continuous, incremental workloads.

A closely related effort to CBP enhances Dryad to automatically identify redundant computation; it caches prior results to avoid re-executing stages or to merge computations with new input [24]. Because these cached results are outside the dataflow, programmers cannot retrieve and store state during execution. CBP takes a different approach, providing programmers explicit access to persistent state through a familiar and powerful groupwise processing abstraction.

Our work also complements recent efforts to build "online" Map-Reduce systems [9]. While their data pipelining techniques for Map-Reduce jobs are orthogonal to the CBP model, the work also describes a controller for running Map-Reduce jobs continuously.




(a) Basic groupwise processing, (b) with access to state, (c) and grouping input with state records.

**Figure 2: The progression from a stateless groupwise processing primitive to stateful translation, $T(\cdot)$, with multiple inputs/outputs, grouped state, and inner groupings.**

The design requires reducers to manage their own internal state, presenting a significant programmer burden as it remains outside of the bulk-processing abstraction. The controller provides limited support for deciding when jobs are runnable and what data they consume. In contrast, CBP dataflow primitives afford a range of policies for controlling these aspects of iterative/incremental dataflows.

Twister [11], a custom Map-Reduce system, optimizes repeatedly run (iterative) Map-Reduce jobs by allowing access to static state. Map and Reduce tasks may persist across iterations, amortizing the cost of loading this static state (e.g., from an input file). However, the state cannot change during iteration. In contrast, CBP provides a general abstraction of state that supports inserts, updates, and removals.

**Data stream management:** CBP occupies a unique place between traditional DBMS and stream processing. Data stream management systems [2] focus on near-real-time processing of continuously-arriving data. This focus leads to an in-memory, record-at-a-time processing paradigm, whereas CBP deals with disk-resident data and set-oriented bulk operations. Lastly, CBP permits cyclic data flows, which are useful in iterative computations and other scenarios described below.

**Incremental view maintenance:** Traditional view-maintenance environments, like data warehousing, use declarative views that are maintained implicitly by the system [3, 25]. In contrast, CBP can be thought of as a platform for generalized view-maintenance; a CBP program is an explicit graph of data transformation steps. Indeed, one can support relational view maintenance on top of our framework, much like relational query languages have been layered on top of Map-Reduce and Dryad (e.g., DryadLINQ [28], Hive [1], Pig [22]).

# 2. STATEFUL BULK PROCESSING

This paper proposes a groupwise processing operator, *translation*, and dataflow primitives to maintain state during continuous bulk data processing. We designed the translate operator to be run repeatedly, allowing users to easily store and retrieve state as new data inputs arrive. The design also enables a range of run-time optimizations for the underlying bulk-processing system. This section first gives an overview of groupwise processing and then describes translation in detail through successive examples. It ends by summarizing the CBP model (Table 1) programmers use to create translation stages.

We choose to add state to a *groupwise* processing construct because it is a core abstraction enabling parallel data processing. Here we use the $reduce : \langle k, v[]\rangle \rightarrow s[]$ function from the Map-Reduce model as our exemplar groupwise processor. It transforms records $v$ grouped by key $k$[1] into zero or more new output records $s$. Groupwise processing underlies many relational and user-defined processing steps. Indeed, upper-layer languages such as Pig [22] and Hive [1] programs compile into a sequence of Map-Reduce jobs, leveraging the inherent data partitioning, sorting, and grouping $reduce$ provides. Equivalently, DryadLINQ [28] and SCOPE [6] compile programs directly into compute DAGs of similar operations on more general dataflow systems like Dryad [14]. Such an interface has proven popular enough not only to warrant its inclusion in these upper-layer languages, but also in commercial databases such as Greenplum, Aster, and Oracle.

## 2.1 Example 1: A basic translate operator

We begin by studying the incremental crawl queue (Figure 1) dataflow in more detail, where each stage is a separate translation operator. We illustrate translate with a simplified version of the *count in-links* stage, called *URLCount*, that only maintains the frequency of observed URLs. This stateful processing stage has a single input that contains URLs extracted from a set of crawled web pages. The output is the set of URLs and counts that changed with the last set of input records.

For illustration, Figure 2 presents a progression from a stateless groupwise primitive, such as $reduce$, to our proposed translate operator, $T(\cdot)$, which will eventually implement *URLCount*. Figure 2(a) shows a single processing *stage* that invokes a user-defined translate function, $T(\cdot)$. To specify the grouping keys, users write a $RouteBy\langle r\rangle$ function that extracts the grouping key from each input record $r$. In the case of *URLCount*, $RouteBy$ extracts the URL as the grouping key. When the groupwise operator executes, the system reads input records, calls $RouteBy$, groups by the key $k$, partitions input data (we illustrate a single partition), and runs operator replicas in parallel for each partition. Each replica then calls $T(\cdot)$ for each grouping key $k$ with the associated records, $r[]$. We call each parallel execution of an operator an *epoch*.

To maintain a frequency count of observed URLs, the $URLCount$ translator needs access to state that persists across epochs. Figure 2(b) adds a logical state module from which a translate function may read or write values for the current grouping key. In our case, *URLCount* stores counts of previously seen URLs, maintaining state records of the type $\{url, count\}$. However, as the next figure shows, translate incorporates state into the grouping operation itself and the semantics of reading and writing to this state module are different than using an external table-based store.

Figure 2(c) shows the full-featured translation function: $T : \langle k, F_S^{in}, F_1^{in}, \ldots, F_n^{in}\rangle$, with multiple logical input and output *flows* and grouped state. As the figure shows, we found it useful

[1]Unlike Map-Reduce, in our model keys are assigned at the entrance to a key-driven operation (e.g., group-by or join), and do not exist outside the context of such an operation.

| Function | Description | Default |
|---|---|---|
| **Translate**(Key,$\Delta F_0^{in}, \ldots, \Delta F_n^{in}$)→ ($\Delta F_0^{out}, \ldots, \Delta F_n^{out}$) | **Per-Stage:** Groupwise transform from input to output records. | — |
| **Runnable**(framingKeys, state) → (reads, removes, state) | **Per-Stage:** Determines if stage can execute and what increments are read/removed. | RunnableALL |
| **FrameBy**(r, state) → (Key, state) | **Per-Flow:** Assign records to input increments. | FrameByPrior |
| **RouteBy**(r) → Key | **Per-Flow:** Extract grouping key from record. | RouteByRcd |
| **OrderBy**(r) → Key | **Per-Flow:** Extract sorting key from record. | OrderByAny |

**Table 1: Five functions control stage processing. Default functions exist for each except for translation.**

```
URLCOUNT_T(url, F^in_state[], F^in_urls[])
1   newcnt ← F^in_urls.size()
2   if F^in_state[0] ≠ NULL then
3       newcnt ← newcnt + F^in_state[0].cnt
4   F^out_state.write({url, newcnt})
5   F^out_updates.write({url, newcnt})
```

**Figure 3: Translator pseudocode that counts observed URLs.**

to model state using explicit, *loopback* flows from a stage output to a stage input. This allows translate to process state records like any other input, and avoids custom user code for managing access to an external store. It also makes it simple for the system to identify and optimize flows that carry state records. For simple stateful translators like $URLCount$ one loopback suffices, $F_S^{out}$ to $F_S^{in}$.

Figure 3 shows pseudocode for our $URLCount$ translate function called within this stage. With multiple logical inputs, it is trivial to separate state from newly arrived records. It counts the number of input records grouped with the given $url$, and writes the updated counts to state and an output flow for downstream stages. A translation stage must explicitly write each state record present in $F_S^{in}$ to $F_S^{out}$ to retain them for the next processing epoch. Thus a translator can discard state records by not propagating them to the output flow. Note that writes are not visible in their groups until the following epoch.

We can optimize the $URLCount$ translator by recognizing that $F_{urls}^{in}$ may update only a fraction of the stored URL counts each epoch. Current bulk-processing primitives provide "full outer" groupings, calling the groupwise function for all found grouping keys. Here $URLCount$ takes advantage of translation's ability to also perform "inner" groupings between state and other inputs. These inner groupings only call translate for state records that have matching keys from other inputs, allowing the system to avoid expensive scans of the entire state flow. However, to improve performance this requires the underlying processing system to be able to randomly read records efficiently (Section 4.5.2).

## 2.2 Example 2: Continuous bulk processing

We now turn our attention to creating more sophisticated translators that either iterate over an input or, in incremental environments, continuously process newly arrived data. A key question for CBP is how to manage continuous data arrivals. For example, an incremental program typically has an external process creating input. CBP systems must decide when to run each stage based on the records accumulating on the input flows. In some cases they may act like existing bulk-processing systems, in which a vertex (a Dryad vertex or a Map-Reduce job) runs when a batch of records exists on each input. They may behave in a manner similar to data stream processors [2], which invoke a dataflow operator when any input has a single tuple available. Or they may behave in some hybrid fashion.

During each processing epoch, the translator, $T(\cdot)$, reads zero

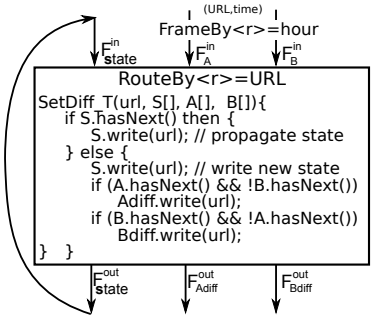



**Figure 4: A stage implementing symmetric set difference of URLs from two input crawls, $A$ and $B$.**

or more records from each input flow, processes them, and writes zero or more records to output flows. Thus a flow $F$ is a sequence of records passed between two processing stages over time. The sequence of records read from a given input flow is called an *input increment*, and a special *input framing* procedure determines the sizes of the input increments. The sequence of records output to a given flow during one epoch form an *output increment*. CBP couples the framing function with a second function, *runnability*, which governs the eligibility of a stage to run (Section 4.2) and also controls consumption of input increments.

We illustrate these concepts by using a CBP program to compare the output of two experimental web crawlers, $A$ and $B$. The stage, illustrated in Figure 4, has an input from each crawler whose records contain (url,timestamp) pairs. Similarly, there is an output for the unique pages found by each crawler. The translator implements symmetric set difference, and we would like to report this difference for each hour spent crawling.[2]

First, the stage should process the same hour of output from both crawlers in an epoch. A CBP stage defines per-flow $FrameBy\langle r\rangle$ functions to help the system determine the input increment membership. The function assigns a *framing key* to each record, allowing the system to place consecutive records with identical framing keys into the same increment. An increment is not eligible to be read until a record with a different key is encountered.[3] Here, $FrameBy$ returns the hour at which the crawler found the URL as the framing key.

However, the stage isn't *runnable* unless we have an hour's worth of crawled URLs on *both* $F_A^{in}$ and $F_B^{in}$. A stage's runnability function has access to the status of its input flows, including the framing keys of each complete increment. The function returns a Boolean value to indicate whether the stage is eligible to run, as well as the

[2]Note that this is the *change* in unique URLs observed; the outputs won't include re-crawled pages (though that is easily done).

[3]The use of *punctuations* [2] can avoid having to wait for a new key, although we have not implemented this feature.

set of flows from which an increment is to be consumed and the set from which an increment is to be removed.

For our symmetric set difference stage, *runnability* returns **true** iff both input flows contain eligible increments. If both input flow increments have the same framing key, the runnability function indicates that both should be read. On the other hand, if the framing keys differ, the runnability function selects only the one with the smaller key to be read. This logic prevents a loss of synchronization in the case that a crawler produces no data for a particular hour.

Finally, the stage's translation function, SetDiff_T, is ready to process observed URLs, storing them in state records. This stage's $RouteBy\langle r\rangle$ function extracts the URL from each input record as the grouping key for state and crawler records. If there is a state record for this url, then it either was reported in a prior epoch or belongs to both crawls (the intersection). In this case the translator only needs to manually propagate the state record. Otherwise, this URL has not been seen and it is written to state. If it was seen exclusively by either crawl, we add it to the appropriate output flow.

Framing and runnability are a powerful combination that allows stages to determine what data to present to a stage, and to synchronize consumption of data across multiple input flows. As with framing functions, runnability functions may maintain a small amount of state. Thus it may contain significant control logic. We have used it to synchronize inputs (e.g., for temporal joins), properly interleave writes to and reads from state, and to maintain static lookup tables (read but not remove an increment). Finally, applications such as PageRank can use it to transition from one iterative phase to another, as we show in Section 3.1.2.

## 2.3 Support for graph algorithms

Groupwise processing supports obvious partitionings of graph problems by assigning a single group to each vertex or edge. For example, programmers can write a single translator that processes all vertices in parallel during each processing epoch. In many cases, those per-vertex translation instances must access state associated with other vertices. To do so, each vertex sends "messages" to other vertices (addressed by their grouping key) so that they may exchange data. Such message passing is a powerful technique for orchestrating large computations (it also underlies Google's graph processing system, Pregel [17]), and the CBP model supports it.

Translation complements message passing in a number of ways. First, using a second loopback flow to carry messages allows an inner grouping with the state used to store the graph. Thus the system will call translate only for the groups representing message destinations. Second, message passing can take advantage of the generality of the $RouteBy$ construct.

Often a computation at a single vertex in the graph affects some or all of the vertices in the graph. For example, our incremental PageRank translator (Section 3.1.2) must broadcast updates of rank from dangling nodes (nodes w/o children) to all other nodes in the graph. Similarly, an update may need to be sent to a subset of the nodes in the graph. While $RouteBy$ can return any number of grouping keys from within a record, there is no simple way for a translator to write a record that includes all nodes in the graph. It is difficult to know the broadcast (or multicast) keyset *a-priori*.

To address this issue, $RouteBy$ supports logical broadcast and multicast grouping keys. Figure 5 shows $RouteBy$ returning the special ALL broadcast key for the input record on $F_1^{in}$. This ensures that the record $b$ becomes associated with all groups found in the input flows. While not shown, it is also possible to limit the broadcast to particular input flows, e.g., only groups found in state. Translators may also associate a subset of grouping keys with a single logical multicast address. Here $RouteBy$ on input flow $F_0^{in}$

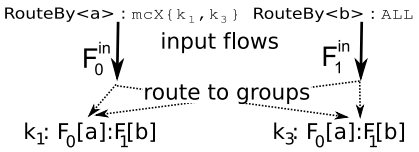



**Figure 5: Users specify per-input flow $RouteBy$ functions to extract keys for grouping. Special keys enable the broadcast and multicast of records to groups. Here we show that multicast address mcX is bound to keys $k_1$ and $k_3$.**

returns a multicast address, mcX, associated with grouping keys $k_1$ and $k_3$. We describe both mechanisms in more detail in Section 4.5.3.

## 2.4 Summary

Naturally, multiple translation stages may be strung together to build more sophisticated incremental programs, such as the incremental crawl queue. In general, a CBP program itself (like Figure 1) is a directed graph $\mathcal{P}$, possibly containing cycles, of translation stages (the vertices), that may be connected with multiple directed flows (the edges). Here we summarize the set of dataflow control primitives in our CBP model that orchestrate the execution of stateful dataflow programs.

As our examples illustrate, CBP controls stage processing through a set of five functions, listed in Table 1. An application may choose these functions, or accept the system-provided defaults (except for translate). The default framing function FrameByPrior returns the epoch number in which the upstream stage produced the record, causing input increments to match output increments generated by upstream stages. The default runnability function, RunnableAll, makes a stage runnable when all inputs have increments and then reads and removes each.

The default $RouteBy$ function, RouteByRcd, gives each record its own group for record-wise processing. Such translators can avoid expensive grouping operations, be pipelined for one-pass execution over the data, and avoid state maintenance overheads. Similarly, the $OrderBy$ function, another key-extraction function that provides per-flow record ordering, has a default OrderByAny, which lets the system select an order that may improve efficiency (e.g., using the order in which the data arrives).

# 3. APPLICATIONS

The collection of default behaviors in the CBP model support a range of important incremental programs, such as the incremental crawl queue example from Section 1.1, which uses RunnableAll and FrameByPrior for all its stages. Here we showcase the extra flexibility the model provides by building stateful, iterative algorithms that operate on graphs.

## 3.1 Mining evolving graphs

Many emerging data mining opportunities operate on large, evolving graphs. Instances of data mining such graphs can be found in systems biology, data network analysis, and recommendation networks in online retail (e.g., Netflix). Here we investigate algorithms that operate over Web and social network graphs. The Web is perhaps the canonical example of a large, evolving graph, and we study an incremental version of the PageRank [5] algorithm used to help index its content. On the other hand, the explosive growth of community sites, such as MySpace or Facebook, have created extremely large social network graphs. For instance, Facebook has over 300 million active users (as of September 2009, see

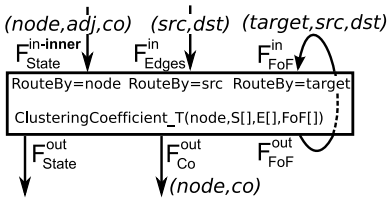



**Figure 6: Incremental clustering coefficient dataflow.**

```
CLUSTERINGCOEFFICIENT_T(node,F^in_state,F^in_edges,F^in_FoF)
1    if F^in_state.hasNext() then state ← F^in_state.next()
2    foreach edge in F^in_edges
3        state.adj.add(edge.dst);
4    foreach edge in F^in_edges
5        foreach target in state.adj
6            F^out_FoF.write(target,edge.src,edge.dst);
7    foreach update in F^in_FoF
8        state.adj[update.src].adj.add(update.dst);
9    if F^in_FoF.hasNext() then
10       recalcCo(state); F^out_Co.write(node,state.co);
11   F^out_state.write(state);
```

**Figure 7: The clustering coefficients translator adds new edges (2-3), sends neighbors updates (4-6), and processes those updates (7-10).**

www.facebook.com/press). These sites analyze the social graph to support day-to-day operations, external querying (Facebook Lexicon), and ad targeting.

### *3.1.1 Clustering coefficients*

We begin with a simple graph analysis, clustering coefficient, that, among other uses, researchers employ to ascertain whether connectivity in social networks reflects real-world trust and relationships [26]. This example illustrates how we load graphs into a stateful processing stage, how to use groupwise processing to iteratively walk across the graph, and how messages may be used to update neighbor's state.

The clustering coefficient of a graph measures how well a graph conforms to the "small-world" network model. A high clustering coefficient implies that nodes form tight cliques with their immediate neighbors. For a node $n_i$, with $N$ neighbors and $E$ edges among the neighbors, the clustering coefficient $c_i = 2E/N(N-1)$.This is simple to calculate if each node has a list of its neighbor's neighbors. In a social network this could be described as a "friends-of-friends" (FoF) relation.

For graph algorithms, we create a grouping key for each unique node in the graph. This allows the calculation to proceed in parallel for each node during an epoch, and us to store state records describing each vertex. Figure 6 illustrates the single stateful stage for incrementally computing clustering coefficients.[4] The input $F^{in}_{edges}$ carries changes to the graph in the form of (src,dst) node ID pairs that represent edges. Records on the state flow reference the node and its clustering coefficient and FoF relation. Each input's $RouteBy$ returns a node ID as the grouping key.

Figure 7 shows the translator pseudocode. The translator must add new graph nodes[5], update adjacency lists, and then update the FoF relations and clustering coefficients. Line 1 retrieves a node's state (an adjacency list, adj, of adjacencies). Each record on $F^{in}_{edges}$ represents a new neighbor for this node. Lines 2-3 add these new neighbors to the local adjacency list. While that code alone is sufficient to build the graph, we must also send these new neighbors to every adjacent node so that they may update their FoF relation.

To do so, we send a record to each adjacent node by writing to the loopback flow $F^{out}_{FoF}$ (lines 4-6). During the next epoch, $RouteBy$ for $F^{in}_{FoF}$ routes these records to the node designated by target. When the system calls *translate* for these nodes, lines 7-10 process records on $F^{in}_{FoF}$, updating the FoF relation and recalculating the clustering coefficient. Finally, line 11 propagates any state changes. Note that the runnability function allows the stage to execute if input is available on *any* input. Thus during one epoch, a translate instance may both incorporate new edges and output new coefficients for prior changes.

There are several important observations. First, it takes two epochs to update the cluster coefficients when the graph changes. This is because "messages" cannot be routed until the following epoch. Second, Figure 6 shows state as an "inner" flow. Thus translation *only* occurs for nodes that have new neighbors (input on $F^{in}_{edges}$) or must update their coefficient (input on $F^{in}_{FoF}$). These two flows actively select the graph nodes for processing each epoch. Finally, where a single input record into the *URLCount* translator causes a single state update, here the work created by adding an edge grows with the size of state. Adding an edge creates messages to update the FoF relation for all the node's neighbors. The message count (and size) grows as the size and connectivity of the graph increase. We explore these implications further in Section 5.3.

[4]Going forward we hide the loop in state loopback flows.

[5]For ease of exposition we do not show edge deletions.

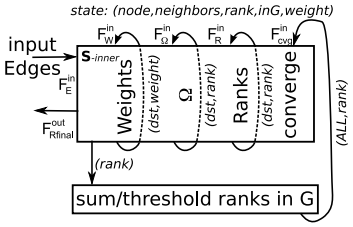



**Figure 8: Incremental PageRank dataflow.**

```
INCRPAGERANK_T(node,F^in_S,F^in_E,F^in_R,F^in_W,F^in_Cvg,F^in_Ω)
1    if F^in_E.hasNext() then makeGraph();startWeight();
2    if F^in_W.hasNext() then sendWeightToNeighbors();
3    if F^in_Ω.hasNext() then updateSupernode();
4    if F^in_Cvg.hasNext() then resetRankState();
5    elseif F^in_R.hasNext() then
6        doPageRankOnG();
```

**Figure 9: Pseudocode for incremental PageRank.**

### *3.1.2 Incremental PageRank*

PageRank is a standard method for determining the relative importance of web pages based on their connectivity [5]. Incremental PageRank is important because (1) computing PageRank on the entire web graph still takes hours on large clusters and (2) important changes to the web graph occur on a small subset of the web (news, blogs, etc.). However, truly incremental PageRank is challenging because small changes (adding a link between pages) can propagate throughout the entire graph. Here we implement the approximate, incremental PageRank computation presented in [8], which thresholds the propagation of PageRank updates. This algorithm takes as input a set of link insertions in the web graph; other approaches exist to incorporate node additions and removals [8].

Figure 8 illustrates our incremental PageRank dataflow, which

shares many features with clustering coefficient. It uses the same format for input edges, groups records by vertex, stores adjacency lists in state records, uses an inner state flow, and sends "messages" to other nodes on loopback flows. We skip the sundry details of translation, and instead focus on how to manage an algorithm that has several distinct iterative phases.

At a high level, the algorithm must build the graph $W$, find the subgraph $G$ affected by newly inserted edges, compute transition probabilities to a supernode $\Omega$ ($W - G$), and then compute PageRank for $G$ (pages in $\Omega$ retain their rank). This algorithm has been shown to be both fast and to provide high-quality approximations for a variety of real and synthesized web crawls [8].

Figure 9 shows high-level pseudocode for the PageRank translator. Internally, the translator acts as a per-node event handler, using the presence of records on each loopback flow as an indication to run a particular phase of the algorithm. Here the *runnability* function plays a critical role in managing phase transitions; it exclusively reads each successive phase's input after the prior input becomes empty. Thus *runnability* first consumes edges from $F^{in}_{edges}$, then $F^{in}_{W}$ (to find $G$), then $F^{in}_{\Omega}$ (updating the supernode), and finally $F^{in}_{R}$ (to begin PageRank on $G$). When doPageRankOnG converges, the second stage writes an ALL record to $F^{out}_{Cvg}$. This causes the translator to reset graph state, readying itself for the next set of edge insertions.

This design attempts to minimize the number of complete scans of the nodes in $W$ by using both "inner" state flows and the multicast ability of the *RouteBy* function. For example, when calculating PageRank for $G$, leaves in $G$ multicast their PageRank to only nodes in $G$. We discuss the multicast API more in Section 4. Finally, note that we place all the phases in a single translator. Other organizations are possible, such as writing a stage for each phase, though this may make multiple copies of the state. In any case, we envision such analytics as just one step in a larger dataflow.

# 4. DESIGN AND IMPLEMENTATION

CBP architectures have two primary layers: dataflow and physical. The physical layer reliably executes and stores the results of a single stage of the dataflow. Above it, the dataflow layer provides reliable execution of an entire CBP dataflow, orchestrating the execution of multiple stages. It ensures reliable, ordered transport of increments between stages and determines which stages are ready for execution. The dataflow layer may also compile the logical dataflow into a more efficient physical representation, depending on the execution capabilities of the physical layer. Such automated analysis and optimization of a CBP dataflow is future work.

## 4.1 Controlling stage inputs and execution

The dataflow layer accepts a CBP dataflow and orchestrates the execution of its multiple stages. The incremental dataflow controller (IDC) determines the set of runnable stages and issues calls to the physical layer to run them.

The IDC maintains a *flow connector*, a piece of run-time state, for each stage's input flow. Each flow connector logically connects an output flow to its destination input flow. It maintains a logical, ordered queue of identifiers that represent the increments available on the associated input flow. Each output flow may have multiple flow connectors, one for each input flow that uses it as a source. After a stage executes, the IDC updates the flow connectors for each output flow by enqueueing the location and *framing* key of each new output increment. The default, with a DefaultFraming *framing* function, is for the stage to produce one output increment per flow per epoch.

The IDC uses a stage's *runnable* function to determine whether a stage can be run. The system passes the function the set of flow connectors with un-read increments and the associated framing keys, and an application-defined piece of state. The *runnable* function has access to each flow connector's meta data (e.g., number of enqueued increments) and determines the set of flow connectors from which to read, *readSet*, and remove, *removeSet*, increments for the next epoch. If the *readSet* is empty, the stage is not runnable. After each epoch, the IDC updates each flow connector, marking increments as read or removing increment references. Increments may be garbage collected when no flow connector references them.

## 4.2 Scheduling with bottleneck detection

The IDC must determine the set of runnable stages and the order in which to run them. Doing so with prior bulk processing systems is relatively straightforward, since they take a DAG as input. In that case a simple on-line topological sort can determine a vertex (stage) execution order that respects data dependencies. However, CBP presents two additional criteria. First, $\mathcal{P}$ may contain cycles, and the scheduler must choose a total order of stages to avoid starvation or high result latency (makespan). Second, using the runnability function, stages can prefer or synchronize processing particular inputs. This means that increments can "back up" on input flows, and that the stage creating data for that input no longer needs to run.

Our simple scheduler executes in phases and may test each stage's *runnability* function. It can detect stage starvation and respond to downstream backpressure (a bottleneck stage) by not running stages that already have increments in all outputs. Full details of this algorithm are available in our techreport [15]).

## 4.3 Failure recovery

The dataflow layer assumes that the physical layer provides atomic execution of individual stages and reliable storage of immutable increments. With such semantics, a single stage may be restarted if the physical layer fails to run a stage. The executed stage specifies a naming convention for each produced increment, requiring it to be tagged by its source stage, flow id, and increment index. These may be encoded in the on-disk path and increment name. Once the physical layer informs the IDC of success, it guarantees that result increments are on disk. Dryad used similar techniques to ensure dataflow correctness under individual job failures [14].

Next, the IDC updates the run-time state of the dataflow. This consists of adding and deleting increment references on existing flow connectors. The controller uses write-ahead logging to record its intended actions; these intentions contain snapshots of the state of the flow connector queue. The log only needs to retain the last intention for each stage. If the IDC fails, it rebuilds state from the XML dataflow description and rebuilds the flow connectors and scheduler state by scanning the intentions.

## 4.4 CBP on top of Map-Reduce

We divide the design and implementation of the CBP model into two parts. In the first part we map $translate$ onto a Map-Reduce model. This is a reasonable starting point for the CBP physical layer due to its data-parallelism and fault-tolerance features. However, this provides an incomplete implementation of the translate operator and CBP dataflow primitives. Further, such a "black-box" emulation results in excess data movement and space usage, sacrificing the promise of incremental dataflows (Section 5). The next section describes our modifications to an open-source Map-Reduce, Hadoop, that supports the full CBP model and optimizes the treatment of state.

The design of our bulk-incremental dataflow engine builds upon the scalability and robustness properties of the GFS/Map-Reduce architecture [13, 10], and in particular the open-source implementation called *Hadoop*. Map-Reduce allows programmers to specify data processing in two phases: map and reduce. The map function outputs a new key-value pair, $\{k_1, v_1\}$, for each input record. The system creates a list of values, $[v]_1$, for each key and passes these to reduce. The Map-Reduce architecture transparently manages the parallel execution of the map phase, the grouping of all values with a given key (the sort), and the parallel execution of the reduce phase.

We now describe how to emulate a single CBP stage using a single Map-Reduce job.[6] Here we describe the Map and Reduce "wrapper" functions that export translate $T(\cdot)$. In CBP applications data is opaque to the processing system, and these wrapper functions encapsulate application data (a record) inside an *application data unit* (ADU) object. The ADU also contains the `flowID`, `RouteByKey`, and `OrderByKey`.

While the Map-Reduce model has one logical input and output, current implementations allow a Map-Reduce job to process multiple input and write multiple output files. In CBP, the `flowID`s within each ADU logically separate flows, and the wrapper code uses the `flowID` to invoke per-flow functions, such as $RouteBy$ and $OrderBy$ that create the routing and ordering keys. This "black-box" approach emulates state as just another input (and output) file of the Map-Reduce job.

• **Map:** The map function wrapper implements routing by running the $RouteBy$ function associated with each input flow. It wraps each input record into an ADU and sets the `flowID`, so the reduce function can separate data originating from the different flows. Map functions may also run one or more *preprocessors* that implement record-wise translation. The optional Map-Reduce combiner has also been wrapped to support distributive or algebraic translators.

• **Reduce:** The Hadoop reducer facility sorts records by the `RouteByKey` embedded in the ADU. Our CBP reduce wrapper function multiplexes the sorted records into $n$ streams, upcalling the user-supplied translator function $T(\cdot)$ with an iterator for each input flow. Per-flow emitter functions route output from $T(\cdot)$ to HDFS file locations specified in the job description. Like the map, emitter functions may also run one or more per-record *postprocessing* steps before writing to HDFS.

Thus a single groupwise translator becomes a job with a map/reduce pair, while a record-wise translator can be a map-only job (allowed by Hadoop) or a reduce postprocessor.

### 4.4.1 Incremental crawl queue example

We illustrate the compilation of a CBP dataflow into Map-Reduce jobs using our incremental crawl queue examples from Figure 1. This dataflow is compiled into two Map-Reduce jobs: CountLinks and DecideCrawl. Figure 10 shows the two jobs and which stages each wrapper function implements. In both jobs all input flows $RouteBy$ the site, and order input by the URL. Otherwise all input flows use the default framing and runnability functions. The first Map-Reduce job implements both *extract links* and *count inlinks*. It writes state ADUs with both site and URL routing keys to maintain counts for each. The second job places both *score* and *threshold* as postprocessing steps on the groupwise *merge* translator. This state flow records all visited src URLs.

[6] An efficient implementation of CBP over a Map-Reduce environment requires deterministic and side-effect-free translators.

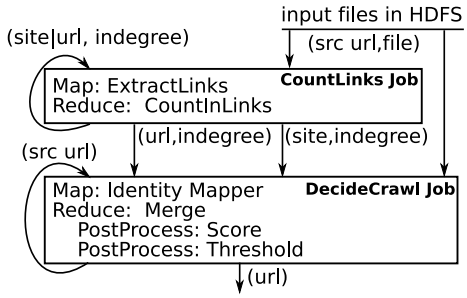



**Figure 10: The Map-Reduce jobs that emulate the CBP incremental crawl queue dataflow.**

### 4.4.2 Increment management

Map-Reduce implementations use shared file systems as a reliable mechanism for distributing data across large clusters. All flow data resides in the Hadoop distributed file system (HDFS). The controller creates a *flow directory* for each flow $F$ and, underneath that, a directory for each increment. This directory contains one or more files containing the ADUs. As discussed in Section 4.2, when Hadoop signals the successful completion of a stage, the controller updates all affected flow connectors.

We emulate custom (non-default) framing functions as post processing steps in the upstream stage whose output flow the downstream stage sources. The reduce wrapper calls the `framing` function for each ADU written to that output flow. By default, the increment directory name is the stage's processing epoch that generated these ADUs. The wrapper appends the resulting `FramingKey` to the increment directory name and writes ADUs with that key to that directory. The wrapper also adds the `FramingKey` to the meta data associated with this increment in the input flow's flow connector. This allows a stage's *runnable* function to compare those keys to synchronize input increments, as described in Section 2.2.

## 4.5 Direct CBP

We now modify Hadoop to accommodate features of the CBP model that are either inexpressible or inefficient as "black-box" Map-Reduce emulations. The first category includes features such as broadcast and multicast record routing. The second category optimizes the execution of bulk-incremental dataflows to ensure that data movement, sorting, and buffering work are proportional to arriving input size, not state size.

### 4.5.1 Incremental shuffling for loopback flows

The system may optimize state flows, and any loopback flow in general, by storing state in per-partition side files. Map-Reduce architectures, like Hadoop, transfer output from each map instance or *task* to the reduce tasks in the *shuffle* phase. Each map task partitions its output into $R$ sets, each containing a subset of the input's grouping keys. The architecture assigns a reduce task to each partition, whose first job is to collect its partition from each mapper.

Hadoop, though, treats state like any other flow, re-mapping and re-shuffling it on each epoch for every groupwise translator. Shuffling is expensive, requiring each reducer to source output from each mapper instance, and state can become large relative to input increments. This represents a large fraction of the processing required to emulate a CBP stage.

However, state is local to a particular translate instance and only contains ADUs assigned to this translate partition. When translators update or propagate existing state ADUs in one epoch, those

ADUs are already in the correct partition for the next epoch. Thus we can avoid re-mapping and re-shuffling these state ADUs. Instead, the reduce task can write and read state from/to an HDFS *partition* file. When a reducer starts, it references the file by partition and merge sorts it with data from the map tasks in the normal fashion.

Note that a translator instance may add state ADUs whose *RouteBy* key belongs to a remote partition during an epoch. These *remote* writes must be shuffled to the correct partition (translation instance) before the next epoch. We accomplish this by simply testing ADUs in the loopback flow's emitter, splitting ADUs into two groups: local and remote. The system shuffles remote ADUs as before, but writes local ADUs to the partition file. We further optimize this process by "pinning" reduce tasks to a physical node that holds a replica of the first HDFS block of the partition file. This avoids reading data from across the network by reading HDFS data stored on the local disk. Finally, the system may periodically re-shuffle the partition files in the case of data skew or a change in processor count.

### 4.5.2 Random access with BIPtables

Here we describe BIPtables (bulk-incremental processing tables), a simple scheme to *index* the state flow and provide random state access to state. This allows the system to optimize the execution of translators that update only a fraction of state. For example, a translator may specify an *inner* state flow, meaning that the system only needs to present state ADUs whose *RouteBy* keys also exist on other inputs. But current bulk-processing architectures are optimized for "streaming" data access, and will read and process inputs in their entirety. This includes direct CBP with state partition files (described above), which reads the entire partition file even if the translator is extremely selective.

However, the success of this approach depends on reading and writing matched keys randomly from a table faster than reading and writing all keys sequentially from a file. Published performance figures for Bigtable, a table-based storage infrastructure [7], indicate a four to ten times reduction in performance for random reads relative to sequential reads from distributed file systems like GFS[13] for 1000-byte records. Moreover, our recent investigation indicates even achieving that performance with open-source versions, such as Hypertable, is optimistic, requiring operations to select under 15% of state keys to improve performance [16]. The design outlined below outperforms sequential when retrieving as many as 60% of the state records (Section 5.2).

BIPtables leverages the fact that our CBP system needs only simple (key, ADUs) retrieval and already partitions and sorts state ADUs, making much of the functionality in existing table-stores redundant or unnecessary. At a high level, each state partition now consists of an *index* and *data* file. While similar to HDFS `MapFile`s or Bigtable's `SSTable` files, they are designed to exist across multiple processing epochs. Logically, the data file is an append-only, unsorted log that contains the state ADUs written over the last $n$ epochs. Because HDFS only supports write-once, non-append files, we create additional HDFS data files each epoch that contain the new state inserts and updates.

Each translate instance reads/writes the entire index file corresponding to its state partition each epoch. They use an in-memory index (like Bigtable) for lookups, and write the index file as a sorted set of key to $\{epoch, offset\}$ pairs. To support inner state flows using BIPtables, we modified reduce tasks to query for state ADUs in parallel with the merge sort of mapper output and to store reads in an ADU cache. This ensures that calls to the translate wrapper do not stall on individual key fetches. Our system learns the set of keys to fetch during the merge and issues reads in parallel. The process ends when the ADU cache fills, limiting the memory footprint, or all keys are fetched. The reduce task probes the ADU cache on each call to the translate wrapper, and misses fault in the offending key.

### 4.5.3 Multicast and broadcast routing

The CBP model extends groupwise processing by supporting a broadcast `ALL` address and dynamic multicast groups. Here we describe how to do so efficiently, reducing duplicate records in the data shuffle. We support `ALL` $RouteBy$ keys by modifying mappers to send `ALL` ADUs to each reduce task during the shuffle phase. At this point, the reduce wrapper will add these tuples to the appropriate destination flow before each call to translate. Since the partition count is often much less than the number of groups in state, this moves considerably less data than shuffling the messages to each group. `ALL` may also specify an optional set of input flows to broadcast to (by default the system broadcasts to all inputs).

While broadcasting has an implicit set of destination keys for each epoch, we provide translator authors the ability to define multicast groups dynamically. They do so by calling $associate(k, mcaddr)$, which associates a target key $k$ with a multicast group $mcaddr$. A translator may call this for any number of keys, making any key a destination for ADUs whose $RouteBy$ returns $mcaddr$. The association and multicast address are only valid for this epoch; the translator must write to this multicast address in the same epoch in which it associates keys.

Under the hood, calls to $associate$ place records of $\{k,mcaddr\}$ on a dynamically instantiated and hidden loopback flow named $f_{mcaddr}$. The system treats input records routed to a multicast address in a similar fashion to `ALL` ADUs, sending a single copy to each reduce task. That record is placed in an in-memory hash table keyed by $mcaddr$. When the reduce wrapper runs, it reads the hidden loopback flow to determine the set of multicast addresses bound to this key and probes the table to retrieve the data.

### 4.5.4 Flow separation in Map-Reduce

While the `FlowID` maintains the logical separation of data in the black-box implementation, the Map-Reduce model and Hadoop implementation treat data from all flows as a single input. Thus the system sorts all input data but must then re-separate it based on `flowID`. It must also order the ADUs on each flow by that flow's *OrderBy* keys. This emulation causes unnecessary comparisons and buffering for groupwise translation.

Consider emulating a groupwise translator with $n$ input flows. A Hadoop reduce tasks calls the reduce function with a single iterator that contains all records (ADUs) sharing a particular key. Direct CBP emulates the individual flow iterators of $T(\cdot)$ by feeding from a single reduce iterator, reading the flow iterators out of `flowID` order forces us to buffer skipped tuples so that they can be read later. A read to that last flow causes the system to buffer the majority of the data, potentially causing OutOfMemoryErrors and aborted processing. This occurs in practice; many of our examples apply updates to state by first reading all ADUs from a particular flow.

We resolve this issue by pushing the concept of a flow into Map-Reduce. Reduce tasks maintain flow separation by associating each mapper with its source input flow. While the number of transfers from the mappers to reducers is unchanged, this reduces the number of primary (and secondary) grouping comparisons on the *RouteBy* (and *OrderBy*) keys. This is a small change to the asymptotic analysis of the merge sort of $r$ records from $m$ mappers from $O(rlogm)$ to $O(rlog\frac{m}{n})$. This speeds up the secondary sort of ADUs sharing

a `RouteByKey` in a similar fashion; the reduce task now employs $n$ secondary sorts based only on the `OrderByKey`. This allows each flow to define its own key space for sorting and permits reading flows in an arbitrary order that avoids unnecessary ADU buffering.

# 5. EVALUATION

Our evaluation establishes the benefits of programming incremental dataflows using the CBP model. It explores how the various optimizations for optimizing data movement improve the performance of our three example programs: the incremental crawl queue, clustering coefficients, and PageRank. We built our CBP prototype using Hadoop version 0.19.1, and the implementation consists of 11k lines of code.

## 5.1 Incremental crawl queue

This part of the evaluation illustrates the benefits of optimizing the treatment of state for incremental programs on a non-trivial cluster and input data set. These experiments use the physical realization of the incremental crawl queue shown in Figure 10. Our input data consists of 27 million web pages that we divide into ten input increments (each appr. 30GB) for the dataflow. We ran our experiments on a cluster of 90 commodity dual core 2.13GHz Xeons with two SATA harddrives and 4GB of memory. The machines have a one gigabit per second Ethernet connection to a shared switch fabric.

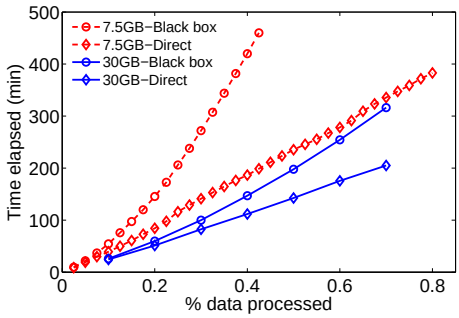



**Figure 11: Cumulative execution time with 30GB and 7.5GB increments.**

The goal of our system is to allow incremental algorithms to achieve per-epoch running times that are a function of the number of state updates, not the total amount of stored state. Note that for the incremental crawl queue, the number of state record updates is directly proportional to the number of arriving input records. Thus, as our test harness feeds the incremental crawl queue successive increments, we expect the running time of each successive increment to be almost constant. To measure the effectiveness of our optimizations, we compare executions of the "black-box" emulation with that of direct CBP.

For some dataflows, including the incremental crawl queue, the benefits of direct CBP increase as increment size decreases. This is because processing in smaller increments forces state flows to be re-shuffled more frequently. Figure 11 shows the cumulative processing time for the black-box and direct systems with two different increment sizes: 30GB (the default) and 7.5GB (dividing the original increment by 4). Though the per-stage running time of direct CBP rises, it still remains roughly linear in the input size (i.e., constant processing time per increment). However, running time using black-box emulation grows super linearly, because the cumulative movement of the state flow slows down processing.

Figure 12 shows a similar experiment using 30GB increments, but reports the individual epoch run times, as well as the run times for the individual CountLinks and DecideCrawl jobs. This experiment includes the strawman, non-incremental processing approach that re-computes the entire crawl queue for each arriving increment. In this case we modify the dataflow so that runs do not read or write state flows. As expected, the running time of the non-incremental dataflow increases linearly, with the majority of the time spent counting in-links. While the incremental dataflow offers a large performance improvement (seen in Figure 12(b)), the runtime still increases with increment count. This is because the black-box emulation pays a large cost to managing the state flow, which continues to grow during the execution of the dataflow. Eventually this reaches 63GB for the *countlinks* stage at the 7th increment.

Figure 12(c) shows run times for the direct CBP implementation that uses incremental shuffling (with reducer pinning) and flow separation. Note that state is an "outer" flow in these experiments, causing translation to access all state ADUs each epoch. Even so, incremental shuffling allows each stage to avoid mapping and shuffling state on each new increment, resulting in a nearly constant runtime. Moreover, HDFS does a good job of keeping the partition file blocks at the prior reducer. At the 7th increment, pinning in direct CBP allows reducers to read 88% of the HDFS state blocks from the local disk.

## 5.2 BIPtable microbenchmarks

These experiments explore whether randomly reading a subset of state is faster using BIPtable than reading all of state sequentially from HDFS. We identify the *break-even* hit rate, the hit rate below which the random access outperforms the sequential access. The test uses a stage that stores a set of unique integers in an inner state flow; input increments contain numbers randomly drawn from the original input. Changing input increment size changes the workload's *hit rate*, the fraction of accessed state. We run the following experiments on a 16-node cluster consisting of Dual Intel Xeon 2.4GHz machines with 4GB of RAM, connected by a Gigabit switch. We pre-loaded the state with 1 million records (500MB). Here translation uses a single data partition, running on a single node, though HDFS (or Hypertable) runs across the cluster.

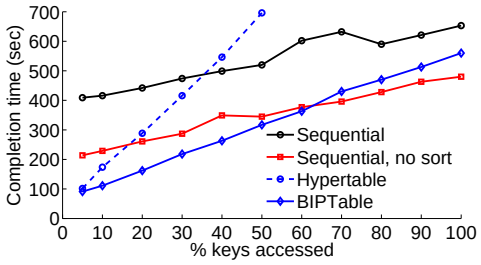
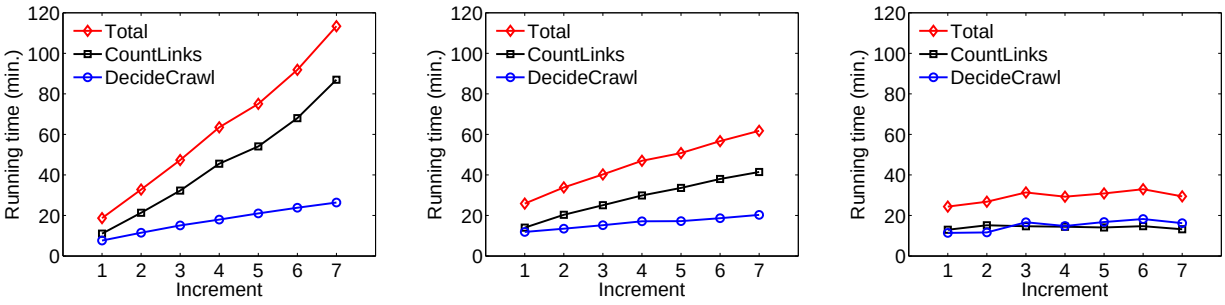



**Figure 13: Running time using indexed state files.**

Figure 13 compares running times for four configurations. *BIPtable* outperforms *Sequential*, which reads the entire state partition file, for every selectivity. One benefit is that *BIPtable* does not sort its records; it uses hashing to match keys on other inputs. To measure this effect, *sequential, no sort* does not sort the partition file (and will therefore incorrectly execute if the translator writes new keys during an epoch). In this case, BIPtable still outperforms sequential access when accessing a majority (>60%) of state. For reference we include a prior result [16] using Hypertable; it failed to produce data when reading more than 50% of state. Finally, it is relatively straightforward for BIPtables to leverage SSDs to im-




(a) Non-incremental dataflow. (b) Incremental dataflow: black-box. (c) Incremental dataflow: direct.

**Figure 12: The performance of the incremental versus landmark crawl queue.**

prove random access performance; a design that promises to significantly extend the performance benefit of this design [16].

## 5.3 Clustering coefficients

Here we explore the performance of our clustering coefficient translator (Figure 7). These graph experiments use a cluster of 25 machines with 160GB drives, 4GB of RAM, and 2.8GHz dual core Xeon processors connected by gigabit Ethernet. We incrementally compute clustering coefficients using a publicly available Facebook crawl [26] that consists of 28 million edges between "friends." We randomize the graph edges and create increments containing 50k edges a piece. These are added to an initial graph of 50k edges connecting 46k vertices.

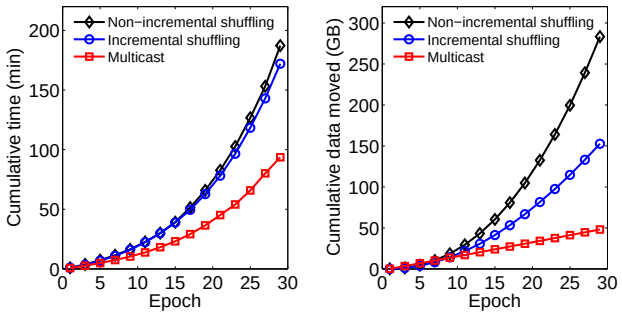
Figure 14(a) shows the cumulative running time for processing successive increments. We configure the translator to use full, outer groupings and successively enable incremental shuffling and multicast support. First note that, unlike the incremental crawl queue, running times with incremental shuffling are not constant. This is because the mapped and shuffled data consists of both messages and state. Recall that these messages must be materialized to disk at the end of the prior epoch and then shuffled to their destination groups during the next epoch. In fact, the message volume increases with each successive increment as the graph becomes increasingly more connected.

Additionally, map tasks that emulate multicasting (i.e, by replicating an input record for each destination) take four to six times as long to execute as map tasks that operate on state records. Hadoop interleaves these longer map tasks with the smaller state map tasks; they act as stragglers until state becomes sufficiently large (around epoch 24). At that point incremental shuffling removes over 50% of the total shuffled data in each epoch, enough to impact running times. Even before then, as Figure 14(b) shows, incremental shuffling frees a significant amount of resources, reducing total data movement by 47% during the course of the experiment.

For this application the critical optimization is multicasting, which both eliminates the user emulating multicast in map tasks and removes duplicate records from the data shuffle. In this case, direct CBP improves cumulative running time by 45% and reduces data shuffled by 84% over the experiment's lifetime.

## 5.4 PageRank

This section explores the impact of direct CBP optimizations on the incremental PageRank dataflow. We have verified that it produces identical results for smaller, 7k node graphs using a non-incremental version. As input we use the "indochina-2004" web graph obtained from [4]; it contains 7.5 million nodes and 109 million edges. These experiments execute on 16 nodes in our cluster




(a) Running time. (b) Data shuffled.

**Figure 14: Incremental clustering coefficient on Facebook data.**

(described above). Here our incremental change is the addition of 2800 random edges (contained in a single input increment).

Figure 15 shows the cumulative execution time for this process. As Section 3.1.2 explained, the dataflow proceeds in three phases: computing PageRank on the original graph (epochs 1-3), finding the subgraph $G$ (epochs 4-8), and re-computing PageRank for nodes in $G$ (epochs 9-16). Here we have purposefully reduced the number of iterations in the first phase to highlight the incremental computation. For this incremental graph update, the affected subgraph $G$ contains 40k nodes.

Here we evaluate the impact of incremental shuffling and inner state flows via BIPtables. Note that this dataflow required the direct CBP implementation, specifically broadcast support for propagating weights from dangling nodes. Without it, local disks filled with intermediate data for even small graphs.

Unlike clustering coefficient, incremental shuffling improves cumulative running time by 23% relative to only using broadcast support. Improvements occur primarily in the last phase as there are fewer messages and processing state dominates. After re-computing PageRank, incremental shuffling has reduced bytes moved by 46%. Finally, we see a significant gain by using inner state flows (BIPtables), as each epoch in the last phase updates only 0.5% of the state records. In this case our architecture reduced both network and CPU usage, ultimately cutting running time by 53%.

# 6. CONCLUSION

A goal of this work is to allow programmers to take advantage of incremental processing in much the same way as prior bulk processing systems have simplified parallel programming. We believe the model strikes a rich balance between sufficient flexibility for

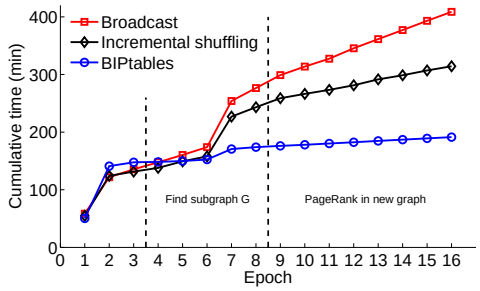



**Figure 15: The cumulative running time of our incremental PageRank translator adding 2800 edges to a 7 million node graph.**

applications and the structure needed by the underlying system to optimize data movement. While the CBP model has relatively few constructs, we recognize that building incremental dataflows by hand is not just tedious but may involve delicate tradeoffs. Thus future work includes providing a compiler to translate an upper-layer language into CBP dataflows, paving the way towards the general study and development of a canon of scalable, incremental algorithms.

## 7. REFERENCES


[1] The Hive project. `http://hadoop.apache.org/hive/`.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of PODS*, March 2002.

[3] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *ACM SIGMOD Record*, 15(2):61–71, 1986.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of WWW'04*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual (web) search engine. 30(7):107–117, 1998.

[6] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of VLDB*, August 2008.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Ch, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI*, pages 205–218, Seattle, WA, November 2006.

[8] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar. Link evolution: Analysis and algorithms. *Internet Mathematics*, 1(3), November 2004.

[9] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In *Proceedings of NSDI*, April 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, San Francisco, CA, December 2004.

[11] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. In *1st International Conference on Cloud Computing (CloudComp09)*, 2009.

[12] M. Ester, H. Kriegal, J. Sander, M. Wimmer, and X. Xu. Incremental clustering and mining in a data warehousing environment. In *Proceedings of VLDB*, 1998.

[13] S. Ghemawat, H. Gogioff, and S. Leung. The Google file system. In *Proceedings of SOSP*, December 2003.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.

[15] D. Logothetis, C. Olston, B. Reed, and K. Yocum. Programming bulk-incremental dataflows. Technical Report CS2009-0944, UCSD, June 2009.

[16] D. Logothetis and K. Yocum. Data indexing for stateful, large-scale data processing. In *Proceedings of NetDB*, October 2009.

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Talk abstract in the Proceedings of PODC*, August 2009.

[18] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of VLDB*, 2002.

[19] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in a spatio-temporal databases. In *Proc. of SIGMOD*, June 2004.

[20] C. Monash. Facebook, Hadoop, and Hive, May 2009. `http://www.dbms2.com/2009/05/11/facebook-hadoop-and-hive`.

[21] H. Ning, W. Xu, Y. Chi, Y. Gong, and T. Huang. Incremental spectral clustering with application to monitoring of evolving blog communities. In *SIAM Int. Conf. on Data Mining*, 2007.

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

[23] B. Pariseau. IDC unstructured data will become the primary task for storage, October 2008. `http://itknowledgeexchange.techtarget.com/storage-soup/idc-unstructured-data-will-become-the-primary-task-for-storage/`.

[24] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc:reusing work in large-scale computations. In *HotCloud Workshop*, June 2009.

[25] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3), September 1991.

[26] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *Proceedings of EuroSys)*, April 2009.

[27] Windows Azure and Facebook teams. Personal communications, August 2008.

[28] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, , and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of OSDI*, San Diego, CA, December 2008.