UC San Diego
Technical Reports

Title
Programming Bulk-Incremental Dataflows

Permalink
https://escholarship.org/uc/item/9bx1c3pk

Authors
Logothetis, Dionysios
Olston, Christofer
Reed, Benjamin
et al.

Publication Date
2009-10-30

Peer reviewed

Programming Bulk-Incremental Dataflows

Dionysios Logothetis
UCSD Computer Science

Christopher Olston
Yahoo! Research

Benjamin Reed
Yahoo! Research

Kevin Webb
UCSD Computer Science

Ken Yocum
UCSD Computer Science

ABSTRACT

Government, medical, financial, and web-based services increasingly depend on the ability to rapidly sift through huge, evolving data sets. These data-intensive applications perform complex multi-step computations over successive generations of data inflows (e.g., weekly web crawls, nightly telescope dumps, or hourly surveillance videos). Because of the data volumes involved, applications must avoid reprocessing old data when new data arrives and instead process incrementally. Unlike in stream-based systems, incoming data does not have to be processed immediately, permitting work to be amortized via bulk processing. Such bulk-incremental processing represents an emerging class of applications whose needs are not fully met by current systems.

This paper presents a generalized architecture for bulk-incremental processing systems (BIPS), simplifying the creation of such programs. In contrast with incremental view maintenance in data warehousing, BIPS provides flexible low-level primitives for managing incremental data and processing, upon which both relational and non-relational operations can be implemented. The paper describes the BIPS programming model along with several example applications and examines some key implementation choices. These choices are shown to play a major role in overall system performance, via experiments on a large testbed cluster.

1. INTRODUCTION

There is growing demand for large-scale processing of loosely structured data, such as text, audio and image files. For example, Internet companies routinely process many terabytes of web documents, user search queries, and other textual data sets. In the image domain, networked surveillance cameras, seismic imaging surveys for oil and gas exploration, and digital medical records (e.g., MRI or X-ray images) can individually account for multiple terabytes of information a day [13]. Other examples include Landsat satellites recording reflected radiation from earth [2], digital sky survey projects [11, 32], and nation-wide mesoscale weather monitoring [9].

The data often has a temporal dimension, with new data arriving periodically in large batches (e.g., weekly web crawls, nightly telescope dumps, hourly satellite passes). Some processing steps operate over data in the current batch only (e.g., correcting for atmospheric distortion; extracting hypertext links), while other steps combine new data with data derived from previous batches (e.g., tracking moving sky objects; maintaining inlink counts). Efficiency dictates that the processing be carried out in an *incremental* fashion.
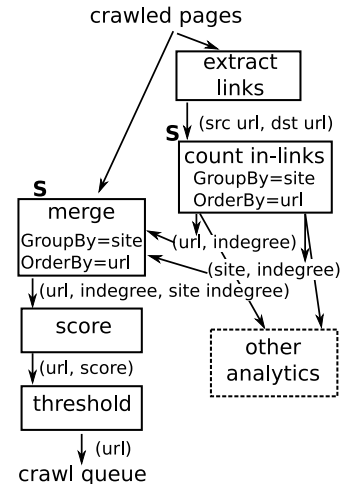

Figure 1: A dataflow for incrementally computing a web crawl queue. Edges represent flows of data, and stages marked with an S are stateful.

Furthermore, many applications favor high throughput over ultra-low latency, so unlike in near-real-time data stream applications, it makes sense to perform *bulk processing* of large data batches. Lastly, these applications involve a significant amount of specialized and non-relational[*]processing, in addition to relational operations.

This paper describes a prototype *bulk-incremental processing system* (BIPS), which offers a simple, scalable and fault-tolerant platform upon which such applications can be deployed. Our work is inspired by non-relational bulk dataflow systems like MapReduce [7] and Dryad [18]. Whereas conventional database systems offer structured relational primitives that can be extended with user-defined data types and operations, Map-Reduce and Dryad aim to impose a minimal amount of structure on data and processing needed to achieve large-scale parallelism and fault-tolerance. BIPS applies the same *minimality principle* to the incremental domain, by adding simple and general support for keeping track of evolving data and processing it in an incremental fashion.

1.1 Example

---

[*]Standard non-relational algorithms that have incremental variants include spatio-temporal queries [20], page rank [8], and data clustering [10, 12, 21], to name a few.

As an illustrative example, we describe a bulk-incremental workflow that computes the crawl queue for a web indexing engine. The algorithm that builds the crawl queue determines the ability of a web index to reflect updates to content within the web. Though crawling is a complex issue [6], Figure 1 gives a high-level overview of a partial, simplified workflow for this process. After bootstrapping with an initial set of pages, the workflow computes a score per URL that is a function of indegree (the number of other pages pointing to this page). Sites that score above some threshold are placed in the crawl queue.

The first processing step or *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.

As in a data stream management environment, each edge transmits new or updated data items, and each stage processes incoming data incrementally. (Stages marked with **S** maintain some state to assist with incremental processing of the inputs.) The *count in-links* stage is an example of a strictly relational processing step, and *extract links* can be thought of as a set-valued UDF. The *merge* stage resembles a relational join, but it is subtly different: it deals with data at two granularities (URLs and sites) simultaneously by grouping by site and performing within-group sorting by URL and outputs tuples that contain both URL-level and site-level data.

1.2 Related Work

1.2.1 Non-Relational Bulk Processing

As mentioned above, our work is inspired by the recent non-relational bulk processing systems Map-Reduce [7] and Dryad [18]. 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 incremental workloads.

1.2.2 Data Stream Management

Data stream management systems [3] focus on near-real-time processing of continuously-arriving temporal data sets. This focus leads to an in-memory, record-at-a-time processing paradigm, whereas BIPS deals with disk-resident data and set-oriented bulk operations. Also, the elastic parallelism underlying BIPS combined with relaxed latency requirements means there is no need for drastic *load shedding* to cope with data spikes. Lastly, BIPS permits cyclic data flows, which are useful in iterative computations and other scenarios described below.

1.2.3 Incremental View Maintenance

BIPS can be thought of as generalized view-maintenance [4, 24, 25] platforms. Indeed, one can imagine supporting 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 [31], Hive [1], Pig [22]).

Unlike traditional view-maintenance environments like data warehousing which use declarative views that are maintained implicitly by the system, BIPS exposes an explicit workflow of data transformation steps. Workflow is a widely-adopted paradigm in many business and scientific domains as a way to manage complex networks of custom data manipulations.

1.2.4 Scientific Workflow

Like BIPS, scientific workflow systems deal with multi-stage data processing flows, and some workflow systems support explicit parallelism. Some scientific workflow systems operate in a grid environment, with the workflow controller orchestrating the movement of data within and between grid sites. BIPS is designed for a single, highly-parallel environment and focuses primarily on support for incremental data and processing.

The only scientific workflow system we are aware of that offers explicit support for incrementality (beyond trivial stateless processing) is [28]. In this system stateful computation is modeled explicitly in the workflow. To facilitate consistent recovery from failures, it restricts state to a particular form: a module's state must consist of a set of unprocessed input tuples and in particular the n most recent ones. In contrast, in our approach state is arbitrary and is controlled by the application. Hence, state can contain processed data (e.g., a running total, or a statistical synopsis of the data seen so far), and is not limited to recent data (e.g., a landmark window).

1.3 Contributions

This paper makes the following contributions:

● **Bulk-incremental data processing model:** BIPS provides a low-level dataflow model for programming bulk-incremental programs. The model supports incremental processing steps, including data-parallel primitives, access to persistent state, and basic synchronization primitives for scheduling processing steps.

● **Efficient implementation:** Existing bulk-processing systems were not developed for incremental processing and leave large opportunities for increment re-use on the table. While simple compile-time optimizations can reduce the impedance mismatch between model and backend execution system, we also identify important modifications that reduce processing overhead. In particular an *incremental shuffle* mechanism reduces unnecessary data movement in the backend system for a wide variety of processing steps.

● **Evaluation:** Experiments with our prototype explore the impact of this paradigm on real-world data and processing tasks. In particular we implement an incremental crawl queue and deploy the prototype across an 88-node cluster on production data. Our main findings are two-fold: (1) A baseline implementation on top of an unmodified Map-Reduce system exhibits unacceptable performance (running time superlinear in input size), whereas our optimized implementation achieves linear running time; (2) Our algorithm for scheduling processing steps responds to back-pressure in one part of a dataflow by avoiding useless work in other parts, thereby streamlining the entire execution.

2. BULK-INCREMENTAL MODEL

This section presents the logical dataflow model of a bulk-incremental processing system. A dataflow program $\mathcal{P}$ is
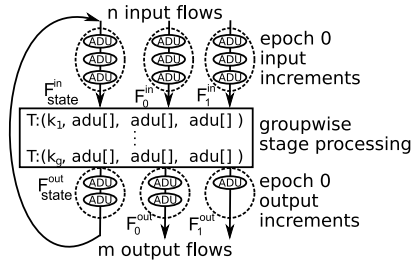
Figure 2: A single logical stage of a BIPS dataflow.

a directed graph, possibly containing cycles, that consists of a set of computation vertices, *stages*, that may be connected with multiple directed edges, *flows*. Each stage may be stateful, supporting incremental processing steps that are distributive, algebraic, or holistic, as defined in [16]. We begin by specifying the structure of data as it flows through the graph, and then describe the processing within a single stage.

2.1 Data model

The core of our data model consists of application data units (ADUs) and flows. Processing stages read and write ADUs; the smallest unit of data in a BIPS. An ADU is a set of bytes with application-defined boundaries; it may be a relational tuple, or any other application-specific data format. A *flow* F is a sequence of ADUs passed between two processing stages over time.

To support a range of application-level semantics, in BIPS, application data is opaque to the processing system. This allows applications to define ADU formats to support their particular brand of incremental computation. For example, each ADU can carry meta data that determines if it is an update, delete, or insert into the stage's state. In BIPS only application-defined functions access the data and instruct the underlying model how to organize and route it.

2.2 Logical stage processing model

A processing stage in a BIPS dataflow has n input flows $\{F_1^{in}, F_2^{in}, \ldots, F_n^{in}\}$, and m output flows $\{F_1^{out}, F_2^{out}, \ldots, F_m^{out}\}$. The stage is invoked repeatedly, in a series of *processing epochs* (epochs, for short). Epochs are local to a stage; there is no notion of global system epochs. In fact, stages may execute concurrently with one another, within the constraints of their eligibility for execution (execution eligibility is discussed in Section 2.6, and scheduling is discussed in Section 4.2).

During one epoch, a stage reads zero or more ADUs from each input flow, and processes those ADUs, and writes zero or more ADUs to output flow. The sequence of ADUs read from a given input flow are called an *input increment*, and the sizes of the input increments are determined via a special *input framing* procedure described below in Section 2.6. The sequence of ADUs output to a given flow during one epoch form an *output increment*. The processing performed by the stage is governed by a *translation function* (translator, for short), $T(\cdot)$, which produces the output increments by applying some (relational or non-relational) transformation over the input increments.

The BIPS model provides three logical facilities upon which the author of a translator may build: state, grouping, and sorting. First, the function $T(\cdot)$ has access to persistent state to support incremental functions. A stage that accesses this state is called *stateful*. We model this state as an implicit, *loopback* flow, adding an input/output flow pair, F_S^{in} and F_S^{out}, to each stateful stage. Figure 2 illustrates these concepts; it shows a single logical BIPS processing stage with $n = m = 2$ and a loopback flow to support state. Each input increment contains three ADUs.

Translation also supports grouping ADUs by application-defined keys. This groupwise processing makes data parallelism a core model feature, allowing bulk data to be processed in parallel for each separate group. Grouping is a fundamental operation underlying many common operations. For example, consider the equi-join operation. If the system groups input data by the predicate's column, a simple translator that iterates over groups can produce the join result.

The model supports grouping via a function $RouteBy\langle ADU \rangle$, which extracts a set of grouping keys from each ADU. There is no restriction on the cardinality of the grouping key set returned for a given ADU. Hence a single ADU may be routed to multiple groups, or to no groups at all, making route-by more general than the standard group-by construct. (We give an example of how this generality can be exploited in Section 2.4.) That said, the most common case is where each ADU resolves to a single group.

In BIPS, groupwise processing subsumes per-ADU processing; stage authors may specify a built-in RouteByADU function to give each ADU its own group. ADU-wise stages allow the BIPS compiler (Section 4.4.1) to make a number of important optimizations. For example, ADU-wise translators can avoid expensive grouping operations, be pipelined together for one-pass execution over the data, and avoid the expense of maintaining state.

It is worth noting that in BIPS grouping is a first-class citizen, decoupled from other data processing logic. By contrast, the Map-Reduce model couples grouping with ADU-wise processing in the "map" function. Map-Reduce programmers routinely use the "map" function to perform general stateless processing (e.g., parsing web pages) in addition to grouping (e.g., grouping web pages by a signature). Such practices harm modularity and often come back to bite. For example, if one wishes to use a different "reduce" function that performs per-website processing, one must modify the web page parsing code to induce grouping by website. In BIPS one simply adds a new $RouteBy$ function, and uses the web-page parsing translator as is.

As Figure 2 shows, the system upcalls the translation function once for each set of ADUs sharing a given grouping key. (Following standard data-parallelism practice, different keys may be handled on different processing nodes.) Note that $T(\cdot)$ also separates the ADUs by flow. Additionally, the BIPS model orders the ADUs in each input flow through another key-extraction function: $OrderBy\langle ADU \rangle$. Such input flow sorting is useful, for example, to write translators for computing order statistics. It is also useful for join-like operations over hierarchical data, as in the *merge* stage of our crawl dataflow example (Section 1.1).

Each input flow to a stage has an associated $RouteBy$ and $OrderBy$ function. The application may choose these functions, or accept the system-provided defaults. The default $RouteBy$ function is RouteByADU. The default $OrderBy$ function is a special OrderByAny function that lets the system select any order (for efficiency reasons the system may

use the order in which the data arrives, but no specific guarantee is made).

2.3 State semantics

Applications "access" state through groupwise translators that present state ADUs that share a given $RouteBy$ key. A stage must explicitly write each state ADU present in F_S^{in} to F_S^{out} to retain them for the next processing epoch. Thus a translator may retain, update, insert or discard, by not writing it to the state output flow, state ADUs during an epoch. In this way, processing epochs are roughly analogous to read/write transactions. Updated and inserted state ADUs do not appear in their groups until the following epoch.

The BIPS model allows translator authors to specify how to group state (on the "left") with respect to the other input flows (on the "right"). By default, the model provides a "full outer" grouping, calling translate for each key present on the state flow, even if no other input flows share that key. While useful for some applications, BIPS allows a stage to specify a "right outer" grouping between state and other inputs, calling translate with only the state ADUs which share keys with ADUs in other input flows. In this case, the model propagates all unseen state tuples automatically.

Our decision to model state as just another input and output flow permits a range of applications. Though it is a single flat list, it accommodates simple coarse-grained state (i.e., state as a black-box entity stored under a single key), or fine-grained hierarchical state (by writing additional ADUs into the state for a particular grouping key, e.g., web sites in our crawler example from Section 1.1). While we considered permitting random access to the state module, we have not found any compelling use cases for it.

2.4 Parameter flows

Typically, stages treat ADUs arriving on input flows as data to be transformed or processed. However, some stages may wish to change their behavior across different processing epochs. BIPS supports this by allowing some input flows to carry stage parameters. For example, a TopK filtering stage specify an input flow holding new data and another input flow holding ADUs that specify k. We call flows used in this manner *parameter* flows, and they differ from data flows in that all ADUs on such a flow must be routed to all groups. Thus k is present when the system invokes the TopK translation function for each group.

The BIPS model allows applications to specify such behavior by using the expressive power of the $RouteBy$ function. The function supports two special symbols: $\emptyset$ and ALL. The $\emptyset$ group indicates that this particular ADU may be discarded. The ALL symbol indicates that this particular ADU should appear on its flow for all found groups.

2.5 Cyclic dataflows

Cycles are a feature that sets the BIPS model apart from many current systems, and they serve a number of important functions. First, they allow a succession of stages to continually refine a data product (iterative computation), integrating program flow control into the dataflow processing system. Second, they allow downstream stages to affect upstream processing through the use of parameter flows. We say that these two kinds of cycles are "real" in the sense that they capture a repeating set of stages within the dataflow.

It is not possible to write a DAG version of the dataflow because the number of iterations is input-dependent.

Cycles can also be used to reduce the state storage overhead within a dataflow. Some dataflows, such as the incremental median application in Section 3, can be written using a single stateful stage. However, that stage both applies state updates (such as entering new numbers) and combines the state with existing downstream output (the median location) to produce some result (the median). These are "faux" cycles in the sense that the dataflow could always be *unfolded* into a DAG that replicates the stateful stage. We discuss this strategy in more detail in Section 3.

2.6 Input Framing and Flow Control

We now turn to the issue of how many ADUs a stage consumes from each input flow in a given epoch, i.e., determination of input increments. By default, a stage's input increments are simply the output increments generated by upstream stages. However, finer-grained control is sometimes needed, e.g., for temporal windowing. It may even be necessary to synchronize consumption of data across multiple input flows, e.g., a temporal join.

To achieve these behaviors, a stage may specify a *framing function*, which assigns a *framing key* to each ADU on each input flow. The framing keys determine the input increments: consecutive ADUs with identical framing keys are placed into the same increment. An increment is not eligible to be read until an ADU with a different key is encountered (the use of *punctuations* [29] can avoid having to wait for a new key, although we have not implemented this feature).

For example, a daily hopping window can be achieved by assigning the framing key to the date associated with a timestamp field inside the ADU. A sliding window of d days would use the same framing function, and have the translator buffer the last d increments in its state. A landmark window would be implemented similarly.

For generality, the framing function has access to some custom state, which is typically small and permits behaviors like windows that are not aligned on pre-determined boundaries like midnight (in that case the state would track the previous window boundary). The default framing function $DefaultFraming$ simply returns the sequence number of the output increment, causing input increments to match output increments as discussed above.

The framing function is coupled with a second function, *runnability*, which governs the eligibility of a stage to run (the issue of scheduling runnable stages is discussed in Section 4.2) and also controls consumption of input increments. The input to the runnability function consists of the framing key of each input flow for which there is a waiting increment. (In some implementations the runnability function may also have access to other meta data, such as the number of increments enqueued on a given input flow.) The function returns a Boolean value to indicate whether the stage is eligible to run, as well as the set of flows from which an increment is to be consumed and the set from which an increment is to be removed. (The decoupling of reading and removal of increments permits behaviors such as static lookup tables.) As with framing functions, runnability functions may maintain a small amount of state.

For example, a stage that performs a temporal join of two input flows by day would use a runnability function that only returns **true** iff both input flows contain eligible

increments. If both input flow increments have the same framing key, the runnability function would indicate that both should be read. On the other hand, if the framing keys differ the runnability function would select only the one with the smaller key to be read. This logic prevents a loss of synchronization in the case that one flow contains no data for a particular day.

The runnability function can be used for other purposes as well, such as preferentially consuming data from one input over another. This functionality is useful if some data paths are more latency-sensitive than others. We will see a concrete example of this scenario in the next section.

The default runnability function, *RunnableAll*, implements the same semantics that Dryad uses to determine stage runnability: the stage is only runnable if all inputs have increments, in which case each increment is both read and removed.

In the case of multiple stages consuming the same flow, framing and runnability are managed separately for each consumer stage. Flow data is only stored once, and each consumer stage maintains its own cursors into the shared flow. Data can be deleted once all consuming flows have performed removal.

3. EXAMPLE APPLICATIONS

We have already described one application of the BIPS model at a high level (the crawl example from Section 1.1). This section presents three other applications that illustrate subtle properties of the model and showcase its flexibility. Keep in mind that, as stated in Section 1, the BIPS model is a *low-level* dataflow programming paradigm that favors flexibility over high-level constructs. Higher-level abstractions (e.g., windowing) and optimization techniques (e.g., state minimization) can be layered on top of BIPS, to make applications more convenient to write and offload the optimization burden from application writers. We leave the design of such layers as future work.

Per-Stage API
translate(Key, $\Delta F_0^{in}, \ldots, \Delta F_n^{in}$)→ ($\Delta F_0^{out}, \ldots, \Delta F_n^{out}$)
runnable(inputFramingKeys, state) → (reads, removes, state)
Per-Flow API
framing(ADU, state) → (Key, state)
routeBy(ADU) → Key
orderBy(ADU) → Key

Table 1: Users define two functions for each translation stage, and two functions for each input flow. Default functions exist for each except for translation.

To summarize the model presented in Section 2, Table 1 lists the five functions that each BIPS dataflow stage implements. The only mandatory one is translate; the remaining ones have defaults. For example, the crawl dataflow (Section 1.1) uses the default runnability function, which waits for all inputs to have available increments, for each stage. When not explicitly mentioned in the following examples, input flows use the default *RouteBy* (RouteByADU), *OrderBy* (OrderByAny), and *framing* (DefaultFraming) functions, as defined in Section 2.2.

3.1 Incremental median

Computing medians is a holistic operation, requiring state in proportion to the input set of integers. We implemented
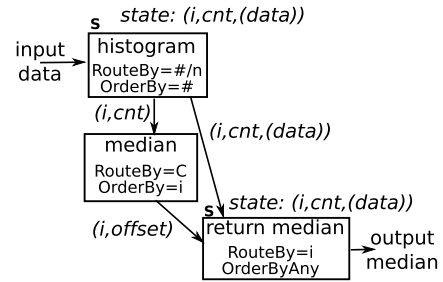
Figure 3: An incremental median BIPS dataflow.
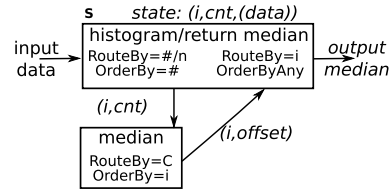
Figure 4: Incremental median with one state copy.

an incremental median in our prototype (Section 4) using the BIPS dataflow shown in Figure 3. The input data, from a finite domain such as $X \in \{0, 1, 2, \ldots, m\}$, first passes through a *RouteBy* function that segments the domain into n buckets. Each parallel instance of the first stage, *histogram*, handles one bucket, maintaining in its state the data and a count of the number of items for that bucket i.[†]Collectively, the counts stored in the state represent a histogram of the input data.

The *histogram* stage passes bucket counts to a second, non-parallel stage (*median*) that determines the bucket and offset of the current median. A final stage (*return median*) uses the median location to query the histogram and locate the value of the median item, which becomes the output of the program. The *return median* stage uses a runnability function that only permits the stage to run when an increment is available on both input flows. This constraint ensures that the stage uses the same version of state that was used to compute the median location.

This simple dataflow constitutes a naive implementation of median, because it maintains two copies of the same state (one in *histogram* and one in *return median*). Moreover, state updates in transit between those two stages represent computational as well as storage overhead. However, cycles in a BIPS dataflow allow us to optimize the median computation by connecting the output of the *median* stage to the *histogram* stage. Figure 4 shows how this cycle eliminates the second state copy. The translate function for the *histogram/median* stage now calculates the last median and updates the state.

In the optimized dataflow, synchronization becomes more subtle. The *histogram/median* stage must wait until the previous histogram's median location arrives, before applying the next increment of state changes. More precisely, it cannot process increment index i on the top input unless increment index $i - 1$ is available on the right input, or $i = 0$.

[†]Moderate skew can be handled in the usual fashion by running a several independent logical parallel instances on each physical processing node.

update tuples

input splitter

(cuboid:group,tuple)

range queries

cube update
RouteBy=
cuboid:group

query
preprocessor

(cuboid:group,aggregates)

(QID,cuboid:group)

state: (cuboid:row,aggregates)

s

data cube
RouteBy=
cuboid:group

(QID,partial answer)

result aggregate
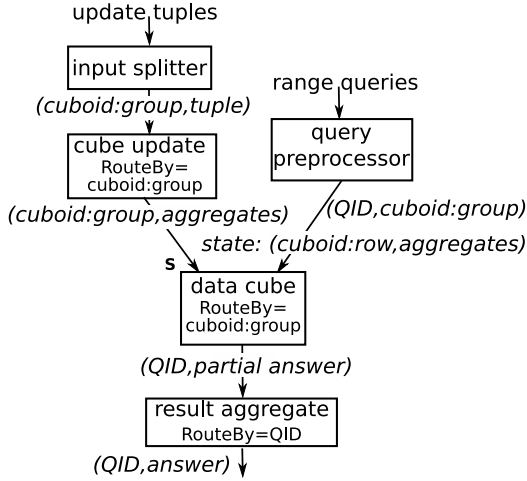RouteBy=QID

(QID,answer)

Figure 5: Incrementally maintaining and querying a data cube as a BIPS dataflow.

This logic resides in the stage's runnability function.

3.2 Incremental data cubing

Here we show how a simple BIPS dataflow can maintain and query a data cube [17]. This example illustrates a case with two external inputs (data to be cubed, and queries to the cube). Our implementation is not meant to compete with existing techniques that change the underlying data representation of the cube to minimize update, storage, and query cost [30, 27, 19]. Rather, it is to show the ease with which we can maintain the raw data cube incrementally leveraging the data-parallelism in the BIPS model.

The dataflow, shown in Figure 5, consists of five stages. The first ADU-wise stage computes updates to the cube. The application interprets input ADUs as 4-tuples; each tuple is an insert into the original table (implementing deletes or updates would require tagging input tuples with the respective operation). An example update tuple would be: $(\mathsf{mon}, \mathsf{10am}, 122, 68)$. The *input split* stage makes 2^n versions of each input tuple, assigning each an extra field indicating the cuboid that it will update. For instance, the tuple $((\mathsf{day}, \mathsf{time}), \mathsf{mon}, \mathsf{10am}, 122, 68)$ will update the $\mathsf{day}, \mathsf{time}$ cuboid.

The *cube update* stage defines a *RouteBy* function for this input flow that returns $(\mathsf{cuboid}, \mathsf{group})$ as the routing key. Using our example tuple, *RouteBy* returns $((\mathsf{day}, \mathsf{time}), (\mathsf{mon}, \mathsf{10am}))$. The stage calculates the aggregate functions for all tuples in each group, with each aggregation function maintaining the necessary state (constant-sized for distributive/algebraic functions, and linear-sized for holistic functions). The stateful groupwise *data cube* stage integrates these updates into each cuboid, allowing updates to proceed in parallel across each cuboid.

Finally, the *query preprocessor* converts range queries into sets of point queries, i.e., queries that return a single row on a particular cuboid. The *result* stage aggregates the answers to the set of point queries representing each range query.

The presence of two external input flows raises a scheduling question. The *data cube* stage may run when an increment is available on either input flow. However, to ensure fast response times for queries to the cube, it may be de-

sirable to configure that stage to process inputs from the query processor preferentially to data updates (with provisions to avoid starving data updates and limit the number of unapplied updates to the data cube). The *data cube* stage's runnability function governs this scheduling decision. Our evaluation (Section 5) uses this dataflow to explore the interaction of runnability and the physical scheduling of stage execution.

3.3 Incremental view maintenance

As alluded to above, general relational view-maintenance [4] techniques can be implemented on top of BIPS. The details are beyond the scope of this paper, but we give a brief sketch here.

In the simplest scenario, we have a relation R and a stream of incoming updates to R. The update stream can be represented in one of two ways. One option is to use two distinct flows: an *insertion* flow and a separate *deletion* flow. Alternatively, one could use a single flow in which the ADUs act as wrappers around the raw tuples and include special markers to indicate whether a given tuple represents insertion, deletion (or perhaps modification, using a key reference). Since BIPS does not impose any particular structure on the ADUs or their interpretation, either approach can be taken.

In either case, a BIPS stage can maintain the base relation R by consuming the update flow(s) along with R, and emitting an updated version of R, such that the flow carrying R forms a self-loop on the stage (obviously the self-loop is not needed in the special case of insert-only with bag semantics). Other stages that want to maintain views over R (including views over R combined with other static or evolving relations, e.g., join views) can subscribe to the output flows that carry R and its updates.

Incremental view maintenance techniques typically apply relational-algebra-like expressions over a combination of base relations, delta relations, and other materialized views (e.g., auxiliary views [26]). With these data elements exposed as flows in a BIPS dataflow, one can implement a view maintenance operation as one or more stages over these flows (e.g., one stage might join R's update flow with the base data flow of another relation S, as part of maintaining the view $R \bowtie S$). Join, set-difference, and other relational-style operations used in view maintenance have efficient out-of-core and parallel implementations based on grouping and sorting [15], which are BIPS primitives.

4. IMPLEMENTATION

Having presented the logical BIPS programming model and examples of its use in Sections 2 and 3, we now turn to implementation. We envision a four-layer "implementation stack" for BIPS systems, shown in Table 2. Each layer groups similar functions together, provides services to the layer above, and receives services from the layer below.

This paper focuses on the design and implementation of the physical and dataflow layers. 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 BIPS 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	Responsibilities
Application	High-level dataflow specification.
Presentation	Libraries and languages.
Dataflow	Optimizes and reliably runs logical dataflow.
Physical	Reliably processes individual stages and stores increments.

Table 2: Bulk-incremental processing designs may be described using four layers.

```
BIPS-SCHEDULER(P, runset)
1      While ! runset.empty()
2          epoch++
3          RUNSTAGE(stage ← runset.rmHead() )
4          stage.runCnt++
5          connectors ← GETNONEMPTYFCS()
6          ForAll fc in connectors
7              if ISRUNNABLE(fc.destStage)
8                  fc.destStage.epochCnt ← epoch
9                  runset.add(fc.destStage)
10         SORTBYRUNBYEPOCH(runset)
```

Figure 6: Pseudocode for stage scheduling.

layer. However, the automated analysis and optimization of a BIPS dataflow into physical jobs is outside the scope of this work.

4.1 Controlling stage inputs and execution

The dataflow layer accepts a BIPS dataflow from the presentation layer 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. We do not address the chal-

lenges of scheduling multiple dataflows, or selecting particular resources for each stage (we leave this to the physical layer). The scheduling decisions made by prior bulk processing systems, such as Dryad, are relatively simple. They take as input a DAG, and use a simple on-line topological sort to determine a vertex (stage) execution order that respects data dependencies.

However, BIPS 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 makespans (time to push a single increment through all stages). 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. We develop a simple scheduler algorithm that avoids stage starvation and that responds to downstream backpressure (a bottleneck stage).

We first describe a scheduler that does not respond to backpressure. It avoids starvation by running every stage at least once before running any stage again. The scheduler has access to all flow connectors and may run each stage's *runnable* function. Scheduling occurs in epochs and determines dependencies dynamically. Each scheduler epoch starts with an ordered set of runnable stages (the *runset*), executes the first, and then determines the new set of runnable stages. We order the stages in the runset by *runCnt*, the number of times the stage has executed, breaking ties by *epochCnt*, the scheduler epoch during which this stage became runnable.

Figure 6 lists pseudocode for our BIPS scheduler. The runset is initially populated with stages that source existing input. The head of the *runset* represents the stage that has run the fewest times, breaking ties by which has been in the *runset* the longest (smallest *epochCnt*). RUNSTAGE asks the physical layer to execute the stage, places produced increments on the flow connectors, and calls the associated framing functions. Next, for each flow connector with input increments we check whether the associated stage is runnable, and, if so, add it to *runset*. Finally, the scheduler re-sorts the *runset*.

Implementing bottleneck detection requires a small change to prune stages that have unread increments on all output flows. Running such stages will not make any downstream stage runnable. This changes line 3 in Figure 6 to find the first stage in the runset that has at least one empty output flow.

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 [18].

Next, the IDC updates the run-time state of the dataflow. This consists of adding new and deleting old increment ref-

erences on to 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 BIPS with Black-box MapReduce

We now turn to implementation options for the BIPS physical layer, which executes runnable stages. In this section we describe a simple implementation strategy on top of an unmodified MapReduce system, which is a reasonable match for BIPS due to its data-parallelism and fault-tolerance features. In Section 4.5 we discuss ways to improve efficiency if one is not limited to MapReduce as an underlying "black-box" system.

Our bulk-incremental dataflow engine leverages the scalability and robustness properties of the GFS/MapReduce architecture [14, 7], and in particular the open-source implementation called *Hadoop*. MapReduce allows programmers to specify data processing in two phases: map and reduce. The map function operates on individual key-value pairs, $\{k_1, v_1\}$, and outputs a new pair, $\{k_2, v_2\}$. The system creates a list of values, $[v]_2$, for each key k_2. The reduce function then creates a final value v_3 from each key-value list pair. The MapReduce 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 BIPS stage using a single MapReduce job.[‡] Here we describe the Map and Reduce "wrapper" functions that export both ADU-wise and groupwise translation functions. These wrapper functions encapsulate BIPS application data inside an ADU object. That object also contains the `flowID`, `RouteByKey`, and `OrderByKey`.

While the MapReduce model has one logical input and output, implementations, GMR or Hadoop, allow a MapReduce job to process multiple input and write multiple output files. In BIPS, 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, and assigns routing keys based on 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*; these preprocessors implement the ADU-wise translation interface. The optional MapReduce combiner has also been wrapped to support applications that provide distributive or algebraic translators.

- **Reduce:** The Hadoop reducer facility sorts records by the `RouteByKey` embedded in the ADU. The BIPS

[‡]To enable an efficient implementation of BIPS on top of a MapReduce environment, we require transformation functions to be deterministic and side-effect-free.
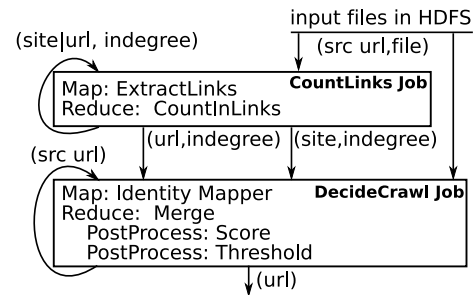

Figure 7: The MapReduce jobs that emulate the BIPS incremental crawl queue dataflow.

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-ADU *postprocessing* steps before writing to HDFS.

Thus a single groupwise translator becomes a job with a map/reduce pair, while an ADU-wise translator can be a map-only job (allowed by Hadoop) or a reduce postprocessor. Note that this presents the dataflow layer with a choice when ADU-wise translators occur sandwiched between two groupwise operations. They may either be postprocessing steps of the first or preprocessing steps of the second. Such choices have also been observed in the Pig Latin compiler, an upper-layer language that also compiles into multiple MapReduce jobs [23].

4.4.1 Incremental crawl queue example

We illustrate the compilation of a BIPS dataflow into MapReduce jobs using our incremental crawl queue examples from Figure 1. This dataflow is compiled into two MapReduce jobs: CountLinks and DecideCrawl. Figure 7 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 MapReduce 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.

4.4.2 Increment management

MapReduce 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 directory for each flow F, called F's *flow directory*, 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 2.6.

4.5 Direct BIPS

While MapReduce architectures support bulk processing and grouping, the emulation of other key BIPS features, such as state flows, remains expensive. This section explores how several aspects of the BIPS model may be *directly* implemented by the physical processing layer to efficiently execute bulk-incremental dataflows. Doing so affords a number of important optimizations to the underlying Hadoop infrastructure for incremental processing: incremental state maintenance and flow separation.

4.5.1 Incremental shuffling for loopback flows

The state flow, and any loopback flow in general, presents special opportunities for optimizing data movement. MapReduce 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 BIPS 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.

In some cases, it is useful to periodically re-shuffle the partition files of a particular loopback flow. For instance, individual partitions may become skewed, allowing some partitions to be much bigger than others. Or a stage may wish to increase or decrease the number of translate instances (re-

duce tasks). We are currently investigating policies for automatically determining when re-shuffling is needed, balancing the one-time cost against future gains in stage throughput.

4.5.2 Flow separation in MapReduce

While the `FlowID` maintains the logical separation of data in the black-box implementation, the MapReduce 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 using a reduce function with a single input. A Hadoop reduce tasks calls the reduce function with a single iterator that contains all records (ADUs) sharing a particular key. The reduce function can take advantage of Hadoop's "secondary sort" to sort this set of ADUs using the `OrderByKey`, with ties broken by `flowID`.[§] The secondary sort presents ADUs with this total ordering using a single iterator, forcing BIPS to emulate the individual flow iterators required by the translator API.

Unfortunately, the black-box emulation requires the programmer to be cognizant of the underlying implementation to avoid significant buffering due to reading flows "out of order." Since this emulation feeds multiple flow iterators from a single reduce iterator, reading the flow iterators out of `flowID` order forces BIPS to buffer any tuples it needs to skip over so that they can be read later. For example, consider the case when ADUs on the last flow have the highest ranking *OrderBy* keys. A read to that last flow will cause 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 stateful stages by first reading all ADUs from a particular flow.

We resolve this issue by pushing the concept of a flow into MapReduce. 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 $O(r log m)$ to $O(r log \frac{m}{n})$. 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 uses a variety of dataflows to establish the benefits of programming incremental dataflows using the BIPS model. It illustrates the benefits of incremental processing by comparing to a non-incremental version of our crawl queue dataflow, and then establishes the importance of directly implementing aspects of the BIPS model in Hadoop. Finally, we show how one can efficiently prioritize bulk processing using the BIPS model for the data cube ap-

---

[§] The original MapReduce work only sorted primary keys.

(a) Landmark dataflow. (b) Incremental dataflow: black-box. (c) Incremental dataflow: direct.
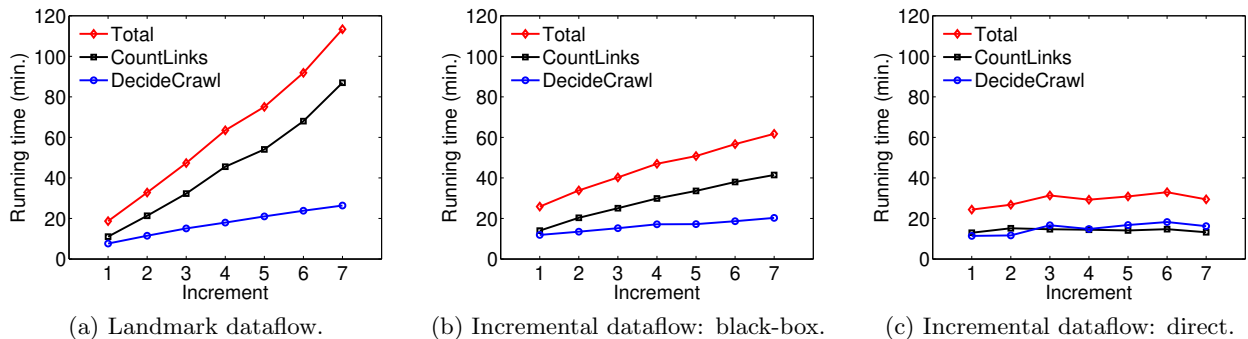
Figure 8: **The performance of the incremental versus landmark crawl queue.**

plication. We built BIPS using Hadoop version 0.17.1, and the implementation consists of 6400 lines of code.

5.1 Incremental crawl queue

This part of the evaluation illustrates the benefits of incremental evaluation on a non-trivial cluster and input data set. These experiments use the physical realization of the incremental crawl queue shown in Figure 7. 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.

To illustrate the benefits of incremental dataflows, we compare the incremental crawl queue to a version that emulates a *landmark* window, where each successive input increment consists of all increments seen so far. This forces the dataflow to re-process all the old data whenever new data arrives.[¶] For the incremental version the system feeds only the new increment to the dataflow. Note that these tests do not "close the loop"; the output crawl queue does not determine the next input increment.

Figure 8 shows the total run times for both landmark and incremental executions of this dataflow, as well as the run times for the individual CountLinks and DecideCrawl jobs. As expected, the running time of the landmark 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 8(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 8(c) shows run times for the direct BIPS implementation that uses incremental shuffling (with reducer pinning) and flow separation. In particular, 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 BIPS allows reducers to read 88% of the HDFS state blocks from the local disk.

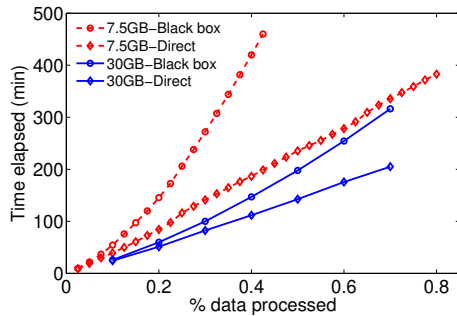[¶]We modify the landmark dataflow so that runs do not read or write state flows.

Figure 9: **Comparing execution cumulative time with 30GB and 7.5GB increments.**

For some dataflows, including the incremental crawl queue, the benefits of direct BIPS increase as increment size decreases. This is because processing in smaller increments forces state flows to be re-shuffled more frequently. Figure 9 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 BIPS 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.

5.2 Bottleneck detection with the data cube

This section explores prioritized flow processing in the data cube dataflow. A data cube may wish to avoid updating the cube under high query workload or wish to provide a lower bound on query throughput. Here our point is to show how one may use BIPS's runnability functions to express prioritized processing. Moreover, such prioritized processing means that certain stages of the dataflow, such as those that create the data updates (*input splitter* and *cube update*), do not have to run as often.

Here each cube update increment contains 5k ADUs, where each ADU is an insert to the cube's base table. Increments to the *query preprocessor* stage consist of 1k ADUs, each representing a different range query and uniformly spread across the cuboids. The runnability function of the *data cube* stage favors query increments over data updates in a 4:1 ratio. For these experiments there are an unbounded
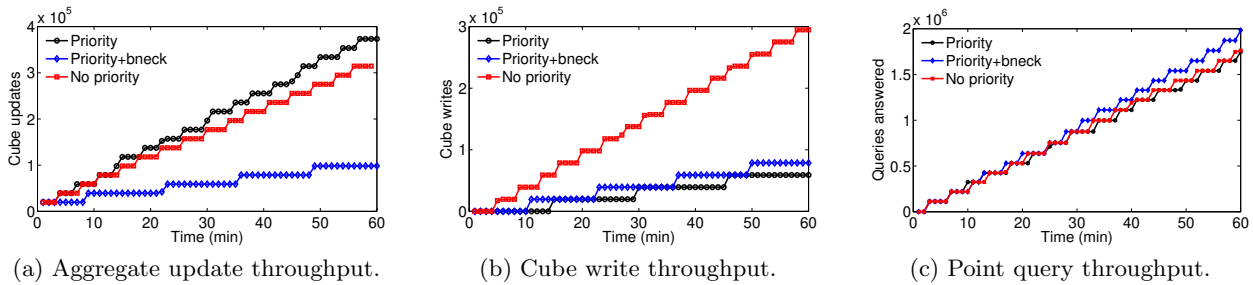
(a) Aggregate update throughput. (b) Cube write throughput. (c) Point query throughput.

Figure 10: Throughput of various stages of the incremental data cube while prioritizing query processing.

number of data update and query increments to process, the data cube runs over a 10 machine cluster connected via gigabit Ethernet, and all results use direct BIPS.

Our experiments investigate the rate at which the dataflow can compute the cube's aggregates (output of *cube update*), write those updates to the cube, and read the cube (output of *result aggregate*). Figure 10 shows a graph for each rate and compares three scenarios. The first, priority+bneck, uses the priority runnability function and the bottleneck-aware scheduler. The second, priority, uses the default scheduler, and the third, no priority, uses the default scheduler and a runnability function that processes any available inputs.

Figure 10(a) shows that the default scheduler runs the *cube update* stage, even if the *data cube* stage no longer needs those inputs (the priority line being much higher than priority+bneck). We can see that the bottleneck-aware scheduler adjusts to the backpressure, and only runs that stage when needed.

Critically, bottleneck-awareness allows the priority runnability function to increase the rate of query processing. Without it, query throughput (Figure 10(c)) is virtually unaffected; the priority and no priority lines lie on top of each other. This is because the default scheduler always runs the *cube update* stage before *data cube*. Having the *data cube* stage process both data updates and queries simultaneously in the no priority experiment does not measurably decrease query throughput, as they can both be done in a single data scan. Using backpressure thus avoids inserting unnecessary delay between query processing steps.

6. CONCLUSION

BIPS provides a flexible computational model that can express a range of continuous, incremental operations, supporting applications from web search, to data mining, to various e-Science data pipelines. While the BIPS model has relatively few constructs, we recognize that building dataflows by hand is tedious. Thus future work includes providing a compiler to translate an upper-layer language into BIPS dataflows.

We have shown that the system model affords an elegant reduction to an existing unstructured processing system, MapReduce. However, experience with this black-box emulation of BIPS revealed impedance mismatches between the model and underlying execution system. A direct implementation of the state flow abstraction, incremental shuffling, allows our BIPS implementation to achieve performance that is nearly proportional to input increment size, not state size. Finally, the BIPS model opens up other promising system

and dataflow-level optimizations, including optimizing right outer groupings by replacing HDFS with a distributed table store (BigTable [5]) and transforming dataflows to automatically reduce the amount of stored state.

Acknowledgements

7. REFERENCES

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

[2] The Landsat Program. http://landsat.gsfc.nasa.gov.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of 21st Symposium on Principles of Database Systems PODS 2002*, March 2002.

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

[5] 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 *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 205–218, Seattle, WA, November 2006.

[6] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins. The discoverability of the web. In *Proceedings of the World Wide Web Conference WWWC*, May 2007.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, San Francisco, CA, December 2004.

[8] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Proceedings of the International World Wide Web Conference WWWC*, May 2005.

[9] K. K. Droegemeier and Co-Authors. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. In *20th Conf. on Interactive Info. Processing Systems for Meteorology, Oceanography, and Hydrology*, 2004.

[10] M. Ester, H. Kriegal, J. Sander, M. Wimmer, and X. Xu. Incremental clustering and mining in a data warehousing environment. In *Proceedings of 24th International Conference on Very Large DataBases (VLDB)*, 1998.

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

[12] V. Ganti, J. Gehrke, and R. Ramakrishnan. DEMON: Mining and monitoring evolving data. *IEEE Transactions on Knowledge and Data Engineering*, 13(1), January 2001.

[13] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe. Technical report, IDC Whitepaper sponsored by EMC Corporation, March 2008.

[14] S. Ghemawat, H. Gogioff, and S. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, December 2003.

[15] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[16] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[17] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, March 2007.

[19] L. V. S. Lakshmanan, J. Pei, and Y. Zhao. Qc-trees: an efficient summary structure for semantic olap. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 64–75, San Diego, California, June 2003. ACM.

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

[21] H. Ning, W. Xu, Y. Chi, Y. Gong, and T. Huang. Incremental spectral clustering with application to monitoring of evolving blog communities. In *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 *SIGMOD'08*, Vancouver, Canada, June 2008.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. of the 2008 ACM SIGMOD Conf. on Management of Data*, June 2008.

[24] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proc. of 28th VLDB*, September 2002.

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

[26] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. Fourth International Conference on Parallel and Distributed Information Systems*, 1996.

[27] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: organization of and bulk incremental updates on the data cube. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 89–99, Tucson, Arizona, USA, June 1997. ACM.

[28] T. Tavares, G. Teodoro, T. Kurc, R. Ferreira, S. Guedes, W. M. Jr., U. Catalyurek, S. Hastings, S. Oster, S. Langella, and J. Saltz. An efficient and reliable scientific workflow system. In *Proc. of 7th International Symposium on Cluster Computing and the Grid (CCGrid'07)*, 2007.

[29] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowledge and Data Engineering*, 2003.

[30] W. Wang, H. Lu, J. Feng, and J. X. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[31] 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 *OSDI'08*, San Diego, CA, December 2008.

[32] 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.