FlumeJava: Easy, Efficient Data-Parallel Pipelines

Craig Chambers, Ashish Raniwala, Frances Perry,
Stephen Adams, Robert R. Henry,
Robert Bradshaw, Nathan Weizenbaum

Google, Inc.
{chambers,raniwala,fjp,sra,rrh,robertwb,nweiz}@google.com

Abstract

MapReduce and similar systems significantly ease the task of writing data-parallel code. However, many real-world computations require a pipeline of MapReduces, and programming and managing such pipelines can be difficult. We present FlumeJava, a Java library that makes it easy to develop, test, and run efficient data-parallel pipelines. At the core of the FlumeJava library are a couple of classes that represent immutable parallel collections, each supporting a modest number of operations for processing them in parallel. Parallel collections and their operations present a simple, high-level, uniform abstraction over different data representations and execution strategies. To enable parallel operations to run efficiently, FlumeJava defers their evaluation, instead internally constructing an execution plan dataflow graph. When the final results of the parallel operations are eventually needed, FlumeJava first optimizes the execution plan, and then executes the optimized operations on appropriate underlying primitives (e.g., MapReduces). The combination of high-level abstractions for parallel data and computation, deferred evaluation and optimization, and efficient parallel primitives yields an easy-to-use system that approaches the efficiency of hand-optimized pipelines. FlumeJava is in active use by hundreds of pipeline developers within Google.

Categories and Subject Descriptors   D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms   Algorithms, Languages, Performance

Keywords   data-parallel programming, MapReduce, Java

1. Introduction

Building programs to process massive amounts of data in parallel can be very hard. MapReduce [6–8] greatly eased this task for data-parallel computations. It presented a simple abstraction to users for how to think about their computation, and it managed many of the difficult low-level tasks, such as distributing and coordinating the parallel work across many machines, and coping robustly with failures of machines, networks, and data. It has been used very successfully in practice by many developers. MapReduce's success in this domain inspired the development of a number of related systems, including Hadoop [2], LINQ/Dryad [20], and Pig [3].

MapReduce works well for computations that can be broken down into a map step, a shuffle step, and a reduce step, but for many real-world computations, a chain of MapReduce stages is required. Such data-parallel *pipelines* require additional coordination code to chain together the separate MapReduce stages, and require additional work to manage the creation and later deletion of the intermediate results between pipeline stages. The logical computation can become obscured by all these low-level coordination details, making it difficult for new developers to understand the computation. Moreover, the division of the pipeline into particular stages becomes "baked in" to the code and difficult to change later if the logical computation needs to evolve.

In this paper we present FlumeJava, a new system that aims to support the development of data-parallel pipelines. FlumeJava is a Java library centered around a few classes that represent *parallel collections*. Parallel collections support a modest number of *parallel operations* which are composed to implement data-parallel computations. An entire pipeline, or even multiple pipelines, can be implemented in a single Java program using the FlumeJava abstractions; there is no need to break up the logical computation into separate programs for each stage.

FlumeJava's parallel collections abstract away the details of how data is represented, including whether the data is represented as an in-memory data structure, as one or more files, or as an external storage service such as a MySql database or a Bigtable [5]. Similarly, FlumeJava's parallel operations abstract away their implementation strategy, such as whether an operation is implemented as a local sequential loop, or as a remote parallel MapReduce invocation, or (in the future) as a query on a database or as a streaming computation. These abstractions enable an entire pipeline to be initially developed and tested on small in-memory test data, running in a single process, and debugged using standard Java IDEs and debuggers, and then run completely unchanged over large production data. They also confer a degree of adaptability of the logical FlumeJava computations as new data storage mechanisms and execution services are developed.

To achieve good performance, FlumeJava internally implements parallel operations using *deferred evaluation*. The invocation of a parallel operation does not actually run the operation, but instead simply records the operation and its arguments in an internal *execution plan* graph structure. Once the execution plan for the whole computation has been constructed, FlumeJava *optimizes* the execution plan, for example *fusing* chains of parallel operations together into a small number of MapReduce operations. FlumeJava then runs the optimized execution plan. When running the execution plan, FlumeJava chooses which strategy to use to implement each operation (e.g., local sequential loop vs. remote parallel MapReduce, based in part on the size of the data being processed), places remote computations near the data they operate on, and per-

forms independent operations in parallel. FlumeJava also manages the creation and clean-up of any intermediate files needed within the computation. The optimized execution plan is typically several times faster than a MapReduce pipeline with the same logical structure, and approaches the performance achievable by an experienced MapReduce programmer writing a hand-optimized chain of MapReduces, but with significantly less effort. The FlumeJava program is also easier to understand and change than the hand-optimized chain of MapReduces.

As of March 2010, FlumeJava has been in use at Google for nearly a year, with 175 different users in the last month and many pipelines running in production. Anecdotal reports are that users find FlumeJava significantly easier to work with than MapReduce.

Our main contributions are the following:

- We have developed a Java library, based on a small set of composable primitives, that is both expressive and convenient.

- We show how this API can be automatically transformed into an efficient execution plan, using deferred evaluation and optimizations such as fusion.

- We have developed a run-time system for executing optimized plans that selects either local or parallel execution automatically and which manages many of the low-level details of running a pipeline.

- We demonstrate through benchmarking that our system is effective at transforming logical computations into efficient programs.

- Our system is in active use by many developers, and has processed petabytes of data.

The next section of this paper gives some background on MapReduce. Section 3 presents the FlumeJava library from the user's point of view. Section 4 describes the FlumeJava optimizer, and Section 5 describes the FlumeJava executor. Section 6 assesses our work, using both usage statistics and benchmark performance results. Section 7 compares our work to related systems. Section 8 concludes.

2. Background on MapReduce

FlumeJava builds on the concepts and abstractions for data-parallel programming introduced by MapReduce. A MapReduce has three phases:

1. The **Map** phase starts by reading a collection of values or key/value pairs from an input source, such as a text file, binary record-oriented file, Bigtable, or MySql database. Large data sets are often represented by multiple, even thousands, of files (called *shards*), and multiple file shards can be read as a single logical input source. The Map phase then invokes a user-defined function, the *Mapper*, on each element, independently and in parallel. For each input element, the user-defined function emits zero or more key/value pairs, which are the outputs of the Map phase. Most MapReduces have a single (possibly sharded) input source and a single Mapper, but in general a single MapReduce can have multiple input sources and associated Mappers.

2. The **Shuffle** phase takes the key/value pairs emitted by the Mappers and groups together all the key/value pairs with the same key. It then outputs each distinct key and a stream of all the values with that key to the next phase.

3. The **Reduce** phase takes the key-grouped data emitted by the Shuffle phase and invokes a user-defined function, the *Reducer*, on each distinct key-and-values group, independently and in parallel. Each Reducer invocation is passed a key and an iterator over all the values associated with that key, and emits zero

or more replacement values to associate with the input key. Oftentimes, the Reducer performs some kind of aggregation over all the values with a given key. For other MapReduces, the Reducer is just the identity function. The key/value pairs emitted from all the Reducer calls are then written to an output sink, e.g., a sharded file, Bigtable, or database.

For Reducers that first combine all the values with a given key using an associative, commutative operation, a separate user-defined *Combiner* function can be specified to perform partial combining of values associated with a given key during the Map phase. Each Map worker will keep a cache of key/value pairs that have been emitted from the Mapper, and strive to combine locally as much as possible before sending the combined key/value pairs on to the Shuffle phase. The Reducer will typically complete the combining step, combining values from different Map workers.

By default, the Shuffle phase sends each key-and-values group to a deterministically but randomly chosen Reduce worker machine; this choice determines which output file shard will hold that key's results. Alternatively, a user-defined *Sharder* function can be specified that selects which Reduce worker machine should receive the group for a given key. A user-defined Sharder can be used to aid in load balancing. It also can be used to sort the output keys into Reduce "buckets," with all the keys of the ith Reduce worker being ordered before all the keys of the $i+1^{st}$ Reduce worker. Since each Reduce worker processes keys in lexicographic order, this kind of Sharder can be used to produce sorted output.

Many physical machines can be used in parallel in each of these three phases.

MapReduce automatically handles the low-level issues of selecting appropriate parallel worker machines, distributing to them the program to run, managing the temporary storage and flow of intermediate data between the three phases, and synchronizing the overall sequencing of the phases. MapReduce also automatically copes with transient failures of machines, networks, and software, which can be a huge and common challenge for distributed programs run over hundreds of machines.

The core of MapReduce is implemented in C++, but libraries exist that allow MapReduce to be invoked from other languages. For example, a Java version of MapReduce is implemented as a JNI veneer on top of the C++ version of MapReduce.

MapReduce provides a framework into which parallel computations are mapped. The Map phase supports embarrassingly parallel, element-wise computations. The Shuffle and Reduce phases support cross-element computations, such as aggregations and grouping. The art of programming using MapReduce mainly involves mapping the logical parallel computation into these basic operations. Many computations can be expressed as a MapReduce, but many others require a sequence or graph of MapReduces. As the complexity of the logical computation grows, the challenge of mapping it into a physical sequence of MapReduces increases. Higher-level concepts such as "count the number of occurrences" or "join tables by key" must be hand-compiled into lower-level MapReduce operations. In addition, the user takes on the additional burdens of writing a driver program to invoke the MapReduces in the proper sequence, managing the creation and deletion of intermediate files holding the data passed between MapReduces, and handling failures across MapReduces.

3. The FlumeJava Library

In this section we present the interface to the FlumeJava library, as seen by the FlumeJava user. The FlumeJava library aims to offer constructs that are close to those found in the user's logical

computation, and abstract away from the lower-level "physical" details of the different kinds of input and output storage formats and the appropriate partitioning of the logical computation into a graph of MapReduces.

3.1 Core Abstractions

The central class of the FlumeJava library is *PCollection<T>*, a (possibly huge) immutable bag of elements of type T. A PCollection can either have a well-defined order (called a *sequence*), or the elements can be unordered (called a *collection*). Because they are less constrained, collections are more efficient to generate and process than sequences. A PCollection<T> can be created from an in-memory Java Collection<T>. A PCollection<T> can also be created by reading a file in one of several possible formats. For example, a text file can be read as a PCollection<String>, and a binary record-oriented file can be read as a PCollection<T>, given a specification of how to decode each binary record into a Java object of type T. Data sets represented by multiple file shards can be read in as a single logical PCollection. For example:[1]

```
PCollection<String> lines =
  readTextFileCollection("/gfs/data/shakes/hamlet.txt");
PCollection<DocInfo> docInfos =
  readRecordFileCollection("/gfs/webdocinfo/part-*",
                           recordsOf(DocInfo.class));
```

In this code, recordsOf(...) specifies a particular way in which a DocInfo instance is encoded as a binary record. Other predefined encoding specifiers are strings() for UTF-8-encoded text, ints() for a variable-length encoding of 32-bit integers, and pairsOf(*e1,e2*) for an encoding of pairs derived from the encodings of the components. Users can specify their own custom encodings.

A second core class is *PTable<K,V>*, which represents a (possibly huge) immutable multi-map with keys of type K and values of type V. PTable<K,V> is a subclass of PCollection<Pair<K,V>>, and indeed is just an unordered bag of pairs. Some FlumeJava operations apply only to PCollections of pairs, and in Java we choose to define a subclass to capture this abstraction; in another language, PTable<K,V> might better be defined as a type synonym of PCollection<Pair<K,V>>.

The main way to manipulate a PCollection is to invoke a data-parallel operation on it. The FlumeJava library defines only a few primitive data-parallel operations; other operations are implemented in terms of these primitives. The core data-parallel primitive is *parallelDo()*, which supports elementwise computation over an input PCollection<T> to produce a new output PCollection<S>. This operation takes as its main argument a DoFn<T, S>, a function-like object defining how to map each value in the input PCollection<T> into zero or more values to appear in the output PCollection<S>. It also takes an indication of the kind of PCollection or PTable to produce as a result. For example:

```
PCollection<String> words =
  lines.parallelDo(new DoFn<String,String>() {
    void process(String line, EmitFn<String> emitFn) {
      for (String word : splitIntoWords(line)) {
        emitFn.emit(word);
      }
    }
  }, collectionOf(strings()));
```

In this code, collectionOf(strings()) specifies that the parallelDo() operation should produce an unordered PCollection whose String elements should be encoded using UTF-8. Other options include sequenceOf(*elemEncoding*)

---

[1] Some of these examples have been simplified in minor ways from the real versions, for clarity and compactness.

for ordered PCollections and tableOf(*keyEncoding, valueEncoding*) for PTables. emitFn is a call-back function FlumeJava passes to the user's process(...) method, which should invoke emitFn.emit(*outElem*) for each *outElem* that should be added to the output PCollection. FlumeJava includes subclasses of DoFn, e.g., MapFn and FilterFn, that provide simpler interfaces in special cases. There is also a version of parallelDo() that allows multiple output PCollections to be produced simultaneously from a single traversal of the input PCollection.

parallelDo() can be used to express both the map and reduce parts of MapReduce. Since they will potentially be distributed remotely and run in parallel, DoFn functions should not access any global mutable state of the enclosing Java program. Ideally, they should be pure functions of their inputs. It is also legal for DoFn objects to maintain local instance variable state, but users should be aware that there may be multiple DoFn replicas operating concurrently with no shared state. These restrictions are shared by MapReduce as well.

A second primitive, *groupByKey()*, converts a multi-map of type PTable<K,V> (which can have many key/value pairs with the same key) into a uni-map of type PTable<K, Collection<V>> where each key maps to an unordered, plain Java Collection of all the values with that key. For example, the following computes a table mapping URLs to the collection of documents that link to them:

```
PTable<URL,DocInfo> backlinks =
  docInfos.parallelDo(new DoFn<DocInfo,
                              Pair<URL,DocInfo>>() {
    void process(DocInfo docInfo,
                 EmitFn<Pair<URL,DocInfo>> emitFn) {
      for (URL targetUrl : docInfo.getLinks()) {
        emitFn.emit(Pair.of(targetUrl, docInfo));
      }
    }
  }, tableOf(recordsOf(URL.class),
             recordsOf(DocInfo.class)));
PTable<URL,Collection<DocInfo>> referringDocInfos =
  backlinks.groupByKey();
```

groupByKey() captures the essence of the shuffle step of MapReduce. There is also a variant that allows specifying a sorting order for the collection of values for each key.

A third primitive, *combineValues()*, takes an input PTable<K, Collection<V>> and an associative combining function on Vs, and returns a PTable<K, V> where each input collection of values has been combined into a single output value. For example:

```
PTable<String,Integer> wordsWithOnes =
  words.parallelDo(
      new DoFn<String, Pair<String,Integer>>() {
    void process(String word,
                 EmitFn<Pair<String,Integer>> emitFn) {
      emitFn.emit(Pair.of(word, 1));
    }
  }, tableOf(strings(), ints()));
PTable<String,Collection<Integer>>
  groupedWordsWithOnes = wordsWithOnes.groupByKey();
PTable<String,Integer> wordCounts =
  groupedWordsWithOnes.combineValues(SUM_INTS);
```

combineValues() is semantically just a special case of parallelDo(), but the associativity of the combining function allows it to be implemented via a combination of a MapReduce combiner (which runs as part of each mapper) and a MapReduce reducer (to finish the combining), which is more efficient than doing all the combining in the reducer.

A fourth primitive, *flatten()*, takes a list of PCollection<T>s and returns a single PCollection<T> that

contains all the elements of the input `PCollections`. `flatten()` does not actually copy the inputs, but rather creates a view of them as one logical `PCollection`.

A pipeline typically concludes with operations that write the final result `PCollections` to external storage. For example:

```
wordCounts.writeToRecordFileTable(
  "/gfs/data/shakes/hamlet-counts.records");
```

Because `PCollections` are regular Java objects, they can be manipulated like other Java objects. In particular, they can be passed into and returned from regular Java methods, and they can be stored in other Java data structures (although they cannot be stored in other `PCollections`). Also, regular Java control flow constructs can be used to define computations involving `PCollections`, including functions, conditionals, and loops. For example:

```
Collection<PCollection<T2>> pcs =
  new Collection<...>();
for (Task task : tasks) {
  PCollection<T1> p1 = ...;
  PCollection<T2> p2;
  if (isFirstKind(task)) {
    p2 = doSomeWork(p1);
  } else {
    p2 = doSomeOtherWork(p1);
  }
  pcs.add(p2);
}
```

3.2   Derived Operations

The FlumeJava library includes a number of other operations on `PCollections`, but these others are derived operations, implemented in terms of these primitives, and no different than helper functions the user could write. For example, the *count()* function takes a `PCollection<T>` and returns a `PTable<T, Integer>` mapping each distinct element of the input `PCollection` to the number of times it occurs. This function is implemented in terms of `parallelDo()`, `groupByKey()`, and `combineValues()`, using the same pattern as was used to compute `wordCounts` above. That code could thus be simplified to the following:

```
PTable<String,Integer> wordCounts = words.count();
```

Another library function, *join()*, implements a kind of join over two or more `PTables` sharing a common key type. When applied to a multi-map `PTable<K, V1>` and a multi-map `PTable<K, V2>`, `join()` returns a uni-map `PTable<K, Tuple2<Collection<V1>, Collection<V2>>>` that maps each key in either of the input tables to the collection of all values with that key in the first table, and the collection of all values with that key in the second table. This resulting table can be processed further to compute a traditional inner- or outer-join, but oftentimes it is more efficient to be able to manipulate the value collections directly without computing their cross-product. `join()` is implemented roughly as follows:

1. Apply `parallelDo()` to each input `PTable<K, Vi>` to convert it into a common format of type `PTable<K, TaggedUnion2<V1,V2>>`.

2. Combine the tables using `flatten()`.

3. Apply `groupByKey()` to the flattened table to produce a `PTable<K, Collection<TaggedUnion2<V1,V2>>>`.

4. Apply `parallelDo()` to the key-grouped table, converting each `Collection<TaggedUnion2<V1,V2>>` into a `Tuple2` of a `Collection<V1>` and a `Collection<V2>`.

Another useful derived operation is *top()*, which takes a comparison function and a count N and returns the greatest N elements of its receiver `PCollection` according to the comparison
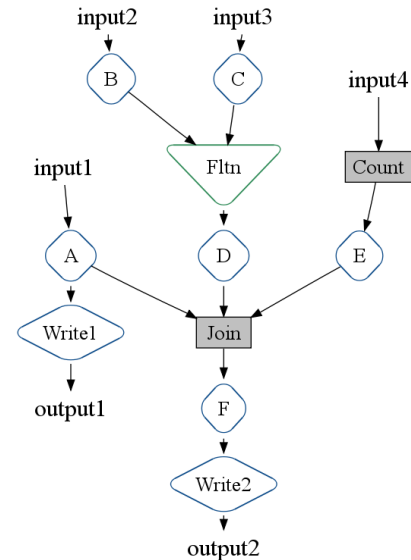

Figure 1. Initial execution plan for the SiteData pipeline.

function. This operation is implemented on top of `parallelDo()`, `groupByKey()`, and `combineValues()`.

The operations mentioned above to read multiple file shards as a single `PCollection` are derived operations too, implemented using `flatten()` and the single-file read primitives.

3.3   Deferred Evaluation

In order to enable optimization as described in the next section, FlumeJava's parallel operations are executed *lazily* using *deferred evaluation*. Each `PCollection` object is represented internally either in *deferred* (not yet computed) or *materialized* (computed) state. A deferred `PCollection` holds a pointer to the *deferred operation* that computes it. A deferred operation, in turn, holds references to the `PCollections` that are its arguments (which may themselves be deferred or materialized) and the deferred `PCollections` that are its results. When a FlumeJava operation like `parallelDo()` is called, it just creates a `ParallelDo` deferred operation object and returns a new deferred `PCollection` that points to it. The result of executing a series of FlumeJava operations is thus a directed acyclic graph of deferred `PCollections` and operations; we call this graph the *execution plan*.

Figure 1 shows a simplified version of the execution plan constructed for the SiteData example used in Section 4.5 when discussing optimizations and in Section 6 as a benchmark. This pipeline takes four different input sources and writes two outputs. (For simplicity, we usually elide `PCollections` from execution plan diagrams.)

- `Input1` is processed by `parallelDo()` A.

- `Input2` is processed by `parallelDo()` B, and `Input3` is processed by `parallelDo()` C. The results of these two operations are `flatten()`ed together and fed into `parallelDo()` D.

- `Input4` is counted using the `count()` derived operation, and the result is further processed by `parallelDo()` E.

- The results of `parallelDo()`s A, D, and E are joined together using the `join()` derived operation. Its result is processed further by `parallelDo()` F.

- Finally, the results of `parallelDo()`s A and F are written to output files.

To actually trigger evaluation of a series of parallel operations, the user follows them with a call to `FlumeJava.run()`. This first optimizes the execution plan and then visits each of the deferred operations in the optimized plan, in forward topological order, and evaluates them. When a deferred operation is evaluated, it converts its result `PCollection` into a materialized state, e.g., as an in-memory data structure or as a reference to a temporary intermediate file. FlumeJava automatically deletes any temporary intermediate files it creates when they are no longer needed by later operations in the execution plan. Section 4 gives details on the optimizer, and Section 5 explains how the optimized execution plan is executed.

3.4 PObjects

To support inspection of the contents of `PCollections` during and after the execution of a pipeline, FlumeJava includes a class `PObject<T>`, which is a container for a single Java object of type T. Like PCollections, PObjects can be either deferred or materialized, allowing them to be computed as results of deferred operations in pipelines. After a pipeline has run, the contents of a now-materialized PObject can be extracted using `getValue()`. PObject thus acts much like a *future* [10].

For example, the `asSequentialCollection()` operation applied to a `PCollection<T>` yields a `PObject<Collection<T>>`, which can be inspected after the pipeline has run to read out all the elements of the computed PCollection as a regular Java in-memory Collection:[2]

```
PTable<String,Integer> wordCounts = ...;
PObject<Collection<Pair<String,Integer>>> result =
  wordCounts.asSequentialCollection();
...
FlumeJava.run();
for (Pair<String,Integer> count : result.getValue()) {
  System.out.print(count.first + ": " + count.second);
}
```

As another example, the `combine()` operation applied to a `PCollection<T>` and a combining function over Ts yields a `PObject<T>` representing the fully combined result. Global sums and maximums can be computed this way.

These features can be used to express a computation that needs to iterate until the computed data converges:

```
PCollection<Data> results =
  computeInitialApproximation();
for (;;) {
  results = computeNextApproximation(results);
  PCollection<Boolean> haveConverged =
    results.parallelDo(checkIfConvergedFn(),
                       collectionOf(booleans()));
  PObject<Boolean> allHaveConverged =
    haveConverged.combine(AND_BOOLS);
  FlumeJava.run();
  if (allHaveConverged.getValue()) break;
}
... continue working with converged results ...
```

The contents of PObjects also can be examined within the execution of a pipeline. One way is using the `operate()` Flume-Java primitive, which takes a list of argument PObjects and an `OperateFn`, and returns a list of result PObjects. When evaluated, `operate()` will extract the contents of its now-materialized argument PObjects, and pass them in to the argument OperateFn. The

OperateFn should return a list of Java objects, which `operate()` wraps inside of PObjects and returns as its results. Using this primitive, arbitrary computations can be embedded within a Flume-Java pipeline and executed in deferred fashion. For example, consider embedding a call to an external service that reads and writes files:

```
// Compute the URLs to crawl:
PCollection<URL> urlsToCrawl = ...;
// Crawl them, via an external service:
PObject<String> fileOfUrlsToCrawl =
  urlsToCrawl.viewAsFile(TEXT);
PObject<String> fileOfCrawledDocs =
  operate(fileOfUrlsToCrawl, new OperateFn() {
    String operate(String fileOfUrlsToCrawl) {
      return crawlUrls(fileOfUrlsToCrawl);
    }
  });
PCollection<DocInfo> docInfos =
  readRecordFileCollection(fileOfCrawledDocs,
                           recordsOf(DocInfo.class));
// Use the crawled documents.
```

This example uses operations for converting between `PCollections` and `PObjects` containing file names. The `viewAsFile()` operation applied to a PCollection and a file format choice yields a `PObject<String>` containing the name of a temporary sharded file of the chosen format where the PCollection's contents may be found during execution of the pipeline. File-reading operations such as `readRecordFileCollection()` are overloaded to allow reading files whose names are contained in PObjects.

In much the same way, the contents of PObjects can also be examined inside a DoFn by passing them in as *side inputs* to `parallelDo()`. When the pipeline is run and the `parallelDo()` operation is eventually evaluated, the contents of any now-materialized PObject side inputs are extracted and provided to the user's DoFn, and then the DoFn is invoked on each element of the input PCollection. For example:

```
PCollection<Integer> values = ...;
PObject<Integer> pMaxValue = values.combine(MAX_INTS);
PCollection<DocInfo> docInfos = ...;
PCollection<Strings> results = docInfos.parallelDo(
    pMaxValue,
    new DoFn<DocInfo,String>() {
      private int maxValue;
      void setSideInputs(Integer maxValue) {
        this.maxValue = maxValue;
      }
      void process(DocInfo docInfo,
                   EmitFn<String> emitFn) {
        ... use docInfo and maxValue ...
      }
    }, collectionOf(strings()));
```

4. Optimizer

The FlumeJava optimizer transforms a user-constructed, modular FlumeJava execution plan into one that can be executed efficiently. The optimizer is written as a series of independent graph transformations.

4.1 `ParallelDo` Fusion

One of the simplest and most intuitive optimizations is `ParallelDo` *producer-consumer fusion*, which is essentially function composition or loop fusion. If one `ParallelDo` operation performs function f, and its result is consumed by another `ParallelDo` operation that performs function g, the two `ParallelDo` operations are replaced by a single multi-output `ParallelDo` that computes both f and $g \circ f$. If the result of the f

---

[2] Of course, `asSequentialCollection()` should be invoked only on relatively small `PCollections` that can fit into memory. FlumeJava includes additional operations such as `asIterable()` that can be used to inspect parts of larger `PCollections`.
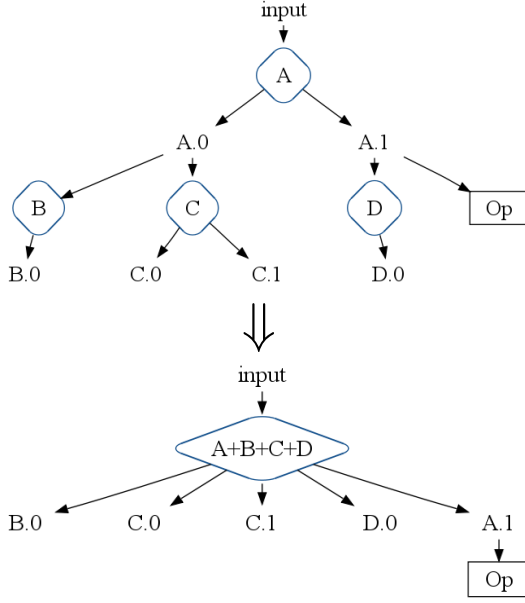
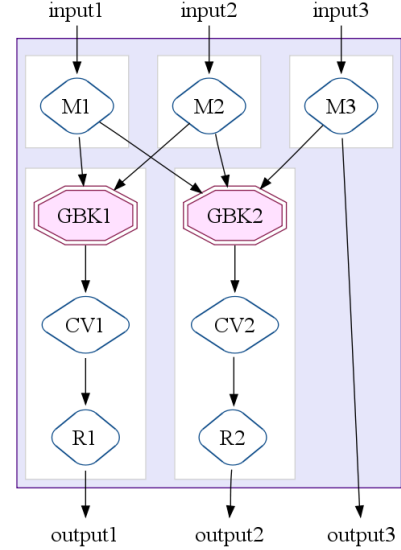**Figure 2.** ParallelDo Producer-Consumer and Sibling Fusion.

Figure 3. A MapShuffleCombineReduce (MSCR) operation with 3 input channels, 2 grouping output channels, and 1 pass-through output channel.

ParallelDo is not needed by other operations in the graph, fusion has rendered it unnecessary, and the code to produce it is removed as dead.

ParallelDo *sibling fusion* applies when two or more ParallelDo operations read the same input PCollection. They are fused into a single multi-output ParallelDo operation that computes the results of all the fused operations in a single pass over the input.

Both producer-consumer and sibling fusion can apply to arbitrary trees of multi-output ParallelDo operations. Figure 2 shows an example execution plan fragment where ParallelDo operations A, B, C, and D can be fused into a single ParallelDo A+B+C+D. The new ParallelDo creates all the leaf outputs from the original graph, plus output A.1, since it is needed by some other non-ParallelDo operation Op. Intermediate output A.0 is no longer needed and is fused away.

As mentioned earlier, CombineValues operations are special cases of ParallelDo operations that can be repeatedly applied to partially computed results. As such, ParallelDo fusion also applies to CombineValues operations.

4.2 The MapShuffleCombineReduce (MSCR) Operation

The core of the FlumeJava optimizer transforms combinations of ParallelDo, GroupByKey, CombineValues, and Flatten operations into single MapReduces. To help bridge the gap between these two abstraction levels, the FlumeJava optimizer includes an intermediate-level operation, the *MapShuffleCombineReduce* (MSCR) operation. An MSCR operation has M input channels (each performing a map operation) and R output channels (each optionally performing a shuffle, an optional combine, and a reduce). Each input channel m takes a PCollection<T_m> as input and performs an R-output ParallelDo "map" operation (which defaults to the identity operation) on that input to produce R outputs of type PTable<K_r,V_r>s; the input channel can choose to emit only to one or a few of its possible output channels. Each output channel r Flattens its M inputs and then either (a) performs a GroupByKey "shuffle", an optional CombineValues "combine", and a O_r-output ParallelDo "reduce" (which de-

faults to the identity operation), and then writes the results to O_r output PCollections, or (b) writes its input directly as its output. The former kind of output channel is called a "grouping" channel, while the latter kind of output channel is called a "pass-through" channel; a pass-through channel allows the output of a mapper to be a result of an MSCR operation.

MSCR generalizes MapReduce by allowing multiple reducers and combiners, by allowing each reducer to produce multiple outputs, by removing the requirement that the reducer must produce outputs with the same key as the reducer input, and by allowing pass-through outputs, thereby making it a better target for our optimizer. Despite its apparent greater expressiveness, each MSCR operation is implemented using a single MapReduce.

Figure 3 shows an MSCR operation with 3 input channels performing ParallelDos M1, M2, and M3 respectively, two grouping output channels, each with a GroupByKey, CombineValues, and reducing ParallelDo, and one pass-through output channel.

4.3 MSCR Fusion

An MSCR operation is produced from a set of *related* GroupByKey operations. GroupByKey operations are considered related if they consume (possibly via Flatten operations) the same input or inputs created by the same (fused) ParallelDo operations.

The MSCR's input and output channels are derived from the related GroupByKey operations and the adjacent operations in the execution plan. Each ParallelDo operation with at least one output consumed by one of the GroupByKey operations (possibly via Flatten operations) is fused into the MSCR, forming a new input channel. Any other inputs to the GroupByKeys also form new input channels with identity mappers. Each of the related GroupByKey operations starts an output channel. If a GroupByKey's result is consumed solely by a CombineValues operation, that operation is fused into the corresponding output channel. Similarly, if the GroupByKey's or fused CombineValues's result is consumed soleby a ParallelDo operation, that operation is also fused into the output channel, if it cannot be fused into a different MSCR's input channel. All the PCollections internal to the fused ParallelDo,
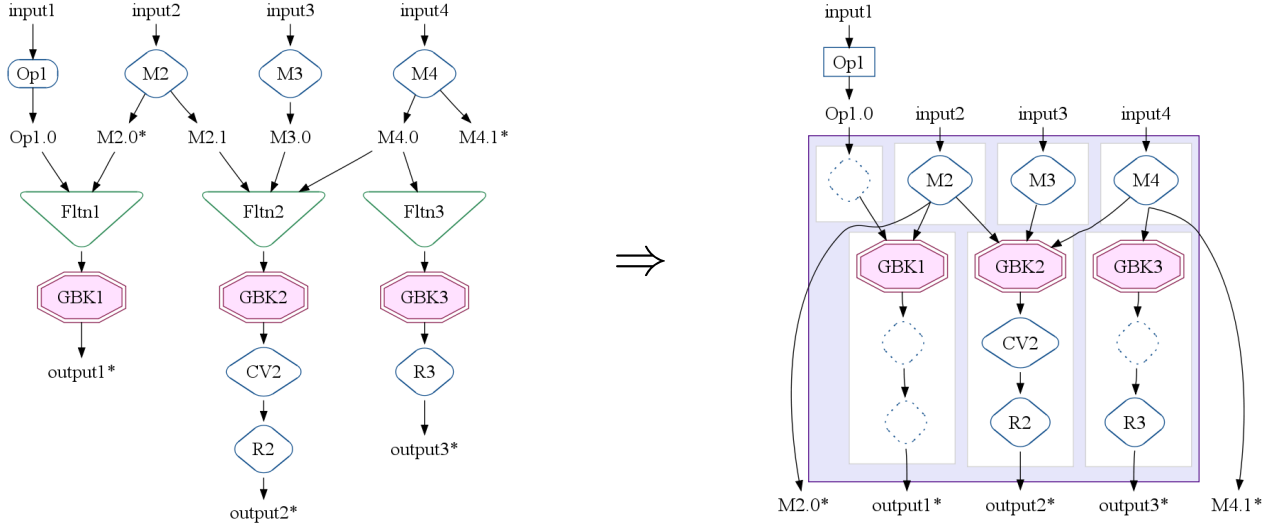
Figure 4. An example of MSCR fusion seeded by three `GroupByKey` operations. Only the starred `PCollections` are needed by later operations.

`GroupByKey`, and `CombineValues` operations are now unnecessary and are deleted. Finally, each output of a mapper `ParallelDo` that flows to an operation or output other than one of the related `GroupByKeys` generates its own pass-through output channel.

Figure 4 shows how an example execution plan is fused into an MSCR operation. In this example, all three `GroupByKey` operations are related, and hence seed a single MSCR operation. GBK1 is related to GBK2 because they both consume outputs of `ParallelDo` M2. GBK2 is related to GBK3 because they both consume PCollection M4.0. The `ParallelDos` M2, M3, and M4 are incorporated as MSCR input channels. Each of the `GroupByKey` operations becomes a grouping output channel. GBK2's output channel incorporates the CV2 `CombineValues` operation. The R2 and R3 `ParallelDos` are also incorporated into output channels. An additional identity input channel is created for the input to GBK1 from non-`ParallelDo` Op1. Two additional pass-through output channels (shown as edges from mappers to outputs) are created for the M2.0 and M4.1 `PCollections` that are used after the MSCR. The resulting MSCR operation has 4 input channels and 5 output channels.

After all `GroupByKey` operations have been transformed into MSCR operations, any remaining `ParallelDo` operations are also transformed into trivial MSCR operations with a single input channel containing the `ParallelDo` and a single pass-through output channel. The final optimized execution plan contains only MSCR, `Flatten`, and `Operate` operations.

4.4 Overall Optimizer Strategy

The optimizer performs a series of passes over the execution plan, with the overall goal to produce the fewest, most efficient MSCR operations in the final optimized plan:

1. *Sink Flattens.* A `Flatten` operation can be pushed down through consuming `ParallelDo` operations by duplicating the `ParallelDo` before each input to the `Flatten`. In symbols, $h(f(a) + g(b))$ is transformed to $h(f(a)) + h(g(b))$. This transformation creates opportunities for `ParallelDo` fusion, e.g., $(h \circ f)(a) + (h \circ g)(b)$.

2. *Lift CombineValues operations.* If a `CombineValues` operation immediately follows a `GroupByKey` operation, the

`GroupByKey` records that fact. The original `CombineValues` is left in place, and is henceforth treated as a normal `ParallelDo` operation and subject to `ParallelDo` fusion.
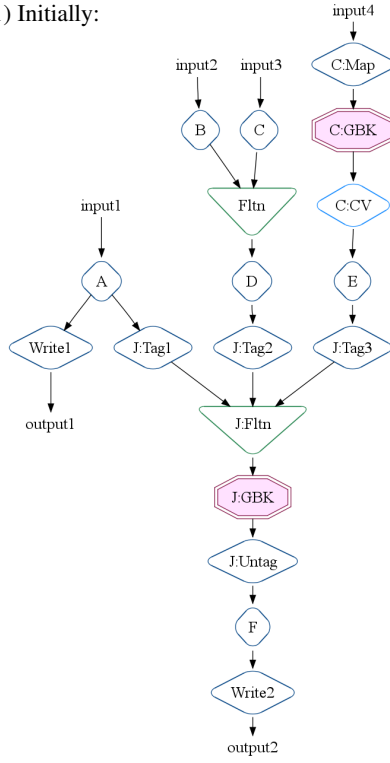
3. *Insert fusion blocks.* If two `GroupByKey` operations are connected by a producer-consumer chain of one or more `ParallelDo` operations, the optimizer must choose which `ParallelDos` should fuse "up" into the output channel of the earlier `GroupByKey`, and which should fuse "down" into the input channel of the later `GroupByKey`. The optimizer estimates the size of the intermediate `PCollections` along the chain of `ParallelDos`, identifies one with minimal expected size, and marks it as boundary blocking `ParallelDo` fusion.

4. *Fuse ParallelDos.*

5. *Fuse MSCRs.* Create MSCR operations. Convert any remaining unfused `ParallelDo` operations into trivial MSCRs.
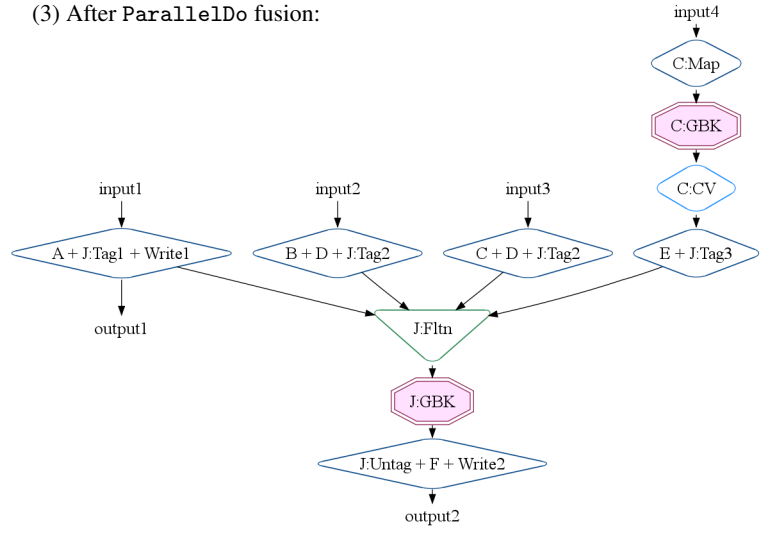
4.5 Example: SiteData

In this section, we show how the optimizer works on the SiteData pipeline introduced in Section 3.3. Figure 5 shows the execution plan initially and after each major optimization phase.

1. *Initially.* The initial execution plan is constructed from calls to primitives like `parallelDo()` and `flatten()` and derived operations like `count()` and `join()` which are themselves implemented by calls to lower-level operations. In this example, the `count()` call expands into `ParallelDo` C:Map, `GroupByKey` C:GBK, and `CombineValues` C:CV, and the `join()` call expands into `ParallelDo` operations J:TagN to tag each of the N input collections, `Flatten` J:Fltn, `GroupByKey` J:GBK, and `ParallelDo` J:Untag to process the results.

2. *After sinking Flattens and lifting CombineValues.* `Flatten` operation Fltn is pushed down through consuming `ParallelDo` operations D and JTag:2. A copy of `CombineValues` operation C:CV is associated with C:GBK.

3. *After ParallelDo fusion.* Both producer-consumer and sibling fusion are applied to adjacent `ParallelDo` operations. Due to fusion blocks, `CombineValues` operation C:CV is not fused with `ParallelDo` operation E+J:Tag3.
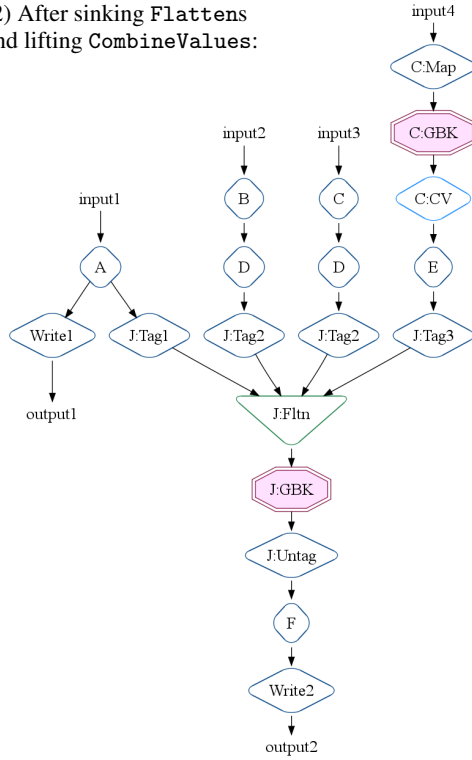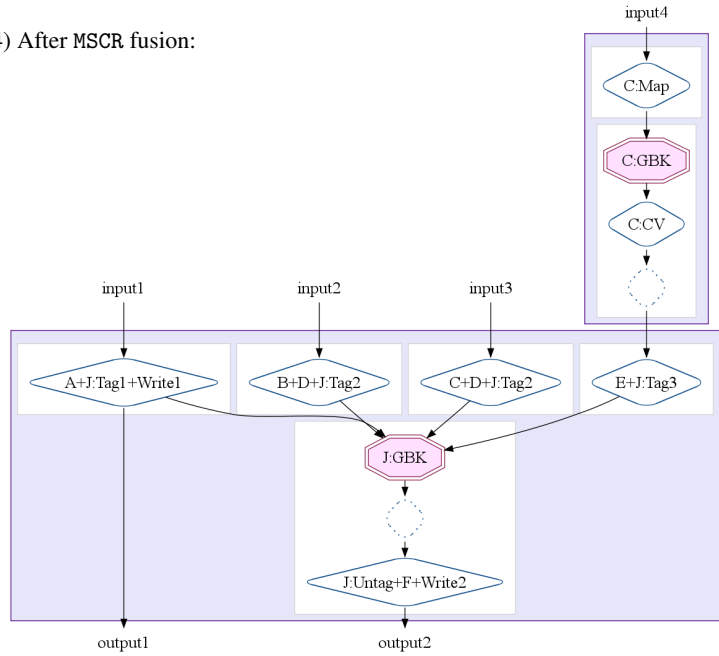
Figure 5. Optimizations applied to the SiteData pipeline to go from 16 original data-parallel operations down to 2 MSCR operations.

4. *After MSCR fusion.* `GroupByKey` operation `C:GBK` and surrounding `ParallelDo` operations are fused into a first `MSCR` operation. `GroupByKey` operations `iGBK` and `J:GBK` become the core operations of a second `MSCR` operation, which includes the remaining `ParallelDo` operations.

The original execution plan had 16 data-parallel operations (`ParallelDos`, `GroupByKeys`, and `CombineValues`). The final, optimized plan has two `MSCR` operations.

4.6 Optimizer Limitations and Future Work

The optimizer does no analysis of the code within user-written functions (e.g., the `DoFn` arguments to `parallelDo()` operations). It bases its optimization decisions on the structure of the execution plan, plus a few optional hints that users can provide giving some information about the behavior of certain operations, such as an estimate of the size of a `DoFn`'s output data relative to the size of its input data. Static analysis of user code might enable better optimization and/or less manual user guidance.

Similarly, the optimizer does not modify any user code as part of its optimizations. For example, it represents the result of fused `DoFn`s via a simple AST-like data structure that explains how to run the user's code. Better performance could be achieved by generating new code to represent the appropriate composition of the user's functions, and then applying traditional optimizations such as inlining to the resulting code.

Users find it so easy to write FlumeJava pipelines that they often write large and sometimes inefficient programs, containing duplicate and/or unnecessary operations. The optimizer could be augmented with additional common-subexpression elimination to avoid duplications. Additionally, users tend to include `groupByKey()` operations more often than necessary, simply because it makes logical sense to them to keep their data grouped by key. The optimizer should be extended to identify and remove unnecessary `groupByKey()` operations, such as when the result of one `groupByKey()` is fed into another (perhaps in the guise of a `join()` operation).

5. Executor

Once the execution plan is optimized, the FlumeJava library runs it. Currently, FlumeJava supports batch execution: FlumeJava traverses the operations in the plan in forward topological order, and executes each one in turn. Independent operations are executed simultaneously, supporting a kind of task parallelism that complements the data parallelism within operations.

The most interesting operation to execute is `MSCR`. FlumeJava first decides whether the operation should be run locally and sequentially, or as a remote, parallel MapReduce. Since there is overhead in launching a remote, parallel job, local evaluation is preferred for modest-size inputs where the gain from parallel processing is outweighed by the start-up overheads. Modest-size data sets are common during development and testing, and by using local, in-process evaluation for these data sets, FlumeJava facilities the use of regular IDEs, debuggers, profilers, and related tools, greatly easing the task of developing programs that include data-parallel computations.

If the input data set appears large, FlumeJava chooses to launch a remote, parallel MapReduce. It uses observations of the input data sizes and estimates of the output data sizes to automatically choose a reasonable number of parallel worker machines. Users can assist in estimating output data sizes, for example by augmenting a `DoFn` with a method that returns the expected ratio of output data size to input data size, based on the computation represented by that `DoFn`. In the future, we would like to refine these estimates through dynamic monitoring and feedback of observed output data sizes, and also to allocate relatively more parallel workers to jobs that have a higher ratio of CPU to I/O.

FlumeJava automatically creates temporary files to hold the outputs of each operation it executes. It automatically deletes these temporary files as soon as they are no longer needed by some unevaluated operation later in the pipeline.

FlumeJava strives to make building and running pipelines feel as similar as possible to running a regular Java program. Using local, sequential evaluation for modest-sized inputs is one way. Another way is by automatically routing any output to `System.out` or `System.err` from within a user's `DoFn`, such as debugging print statements, from the corresponding remote MapReduce worker to the main FlumeJava program's output streams. Likewise, any exceptions thrown within a `DoFn` running on a remote MapReduce worker are captured, sent to the main FlumeJava program, and rethrown.

When developing a large pipeline, it can be time-consuming to find a bug in a late pipeline stage, fix the program, and then reexecute the revised pipeline from scratch, particularly when it is not possible to debug the pipeline on small-size data sets. To aid in this cyclic process, the FlumeJava library supports a *cached* execution mode. In this mode, rather than recompute an operation, FlumeJava first attempts to reuse the result of that operation from the previous run, if it was saved in a (internal or user-visible) file and if FlumeJava determines that the operation's result has not changed. An operation's result is considered to be unchanged if (a) the operation's inputs have not changed, and (b) the operation's code and captured state have not changed. FlumeJava performs an automatic, conservative analysis to identify when reuse of previous results is guaranteed to be safe; the user can direct additional previous results to be reused. Caching can lead to quick edit-compile-run-debug cycles, even for pipelines that would normally take hours to run.

FlumeJava currently implements a batch evaluation strategy, for a single pipeline at a time. In the future, it would be interesting to experiment with a more incremental, streaming, or continuous execution of pipelines, where incrementally added input leads to quick, incremental update of outputs. It also would be interesting to investigate optimization across pipelines run by multiple users over common data sources.

6. Evaluation

We have implemented the FlumeJava library, optimizer, and executor, building on MapReduce and other lower-level services available at Google.

In this section, we present information about how FlumeJava has been used in practice, and demonstrate experimentally that the FlumeJava optimizer and executor make modular, clear FlumeJava programs run nearly as well as their hand-optimized raw-MapReduce-based equivalents.

6.1 User Adoption and Experience

One measure of the utility of the FlumeJava system is the extent to which real developers find it worth converting to from systems they already know and are using. This is the principal way in which we evaluate the FlumeJava programming abstractions and API.

Since its initial release in May 2009, FlumeJava has seen significant user adoption and production use within Google. To measure usage, we instrumented the FlumeJava library to log a usage record every time a FlumeJava program is run. The following table presents some statistics derived from these logs, as of mid-March 2010:[3]

---

[3] The FlumeJava usage logs themselves are processed using a FlumeJava program.

1-day active users	62
7-day active users	106
30-day active users	176
Total users	319

The N-day active users numbers give the number of distinct user ids that ran a FlumeJava program (excluding canned tutorial programs) in the previous N days.

Hundreds of FlumeJava programs have been written and checked in to Google's internal source-code repository. Individual FlumeJava programs have been run successfully on thousands of machines over petabytes of data.

In general, users seem to be very happy with the FlumeJava abstractions. They are not always as happy with some aspects of Java or FlumeJava's use of Java. In particular, Java provides poor support for simple anonymous functions and heterogeneous tuples, which leads to verbosity and some loss of static type safety. Also, FlumeJava's `PCollection`-based data-parallel model hides many of the details of the individual parallel worker machines and the subtle differences between Mappers and Reducers, which makes it difficult to express certain low-level parallel-programming techniques used by some advanced MapReduce users.

FlumeJava is now slated to become the primary Java-based API for data-parallel computation at Google.

6.2 Optimizer Effectiveness

In order to study the effectiveness of the FlumeJava optimizer at reducing the number of parallel MapReduce stages, we instrumented the FlumeJava system so that it logs the structure of the user's pipeline, before and after optimization. The scatterplot below shows the results extracted from these logs. Each point in the plot depicts one or more user pipelines with the corresponding number of stages. To aid the readability of the plot, we removed data on about 10 larger pipelines with more than 120 unoptimized stages.
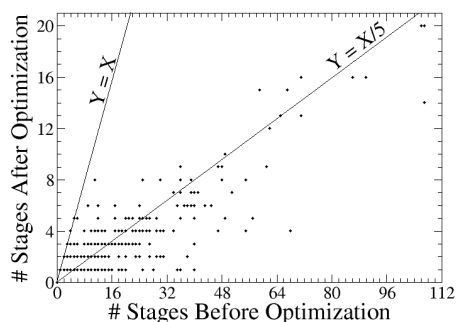
Looking at the sizes of the user pipelines both before and after optimization, it is evident that FlumeJava has been used for writing small as well as fairly large pipelines. In fact, the largest of the pipeline so far (not plotted) had 820 unoptimized stages and 149 optimized stages. This data further underscores the usability of the FlumeJava API.

Looking at the optimizer's "compression" ratio (the ratio of number of stages before and after the optimization), the optimizer appears to achieve on average a 5x reduction in the number of stages, and some pipelines had compression ratios over 30x. One pipeline (not plotted) had 207 unoptimized stages which were fused into a single optimized stage.

The FlumeJava optimizer itself runs quickly, especially compared to the actual execution that follows optimization. For pipelines having up to dozens of operations, the optimizer takes less than a second or two.

6.3 Execution Performance

The goal of FlumeJava is to allow a programmer to express his or her data-parallel computation in a clear, modular way, while simultaneously executing it with performance approaching that of the best possible hand-optimized programs written directly against MapReduce APIs. While high optimizer compression is good, the real goal is small execution time.

To assess how well FlumeJava achieves this goal, we first constructed several benchmark programs, based on real pipelines written by FlumeJava users. These benchmarks performed different computational tasks, including analyzing ads logs (Ads Logs), extracting and joining data about websites from various sources (SiteData and IndexStats), and computing usage statistics from logs dumped by internal build tools (Build Logs).

We wrote each benchmark in three different ways:

- in a modular style using FlumeJava,
- in a modular style using Java MapReduce, and
- in a hand-optimized style using Java MapReduce.

For two of the benchmarks, we also wrote in a fourth way:

- in a hand-optimized style using Sawzall [17], a domain-specific logs-processing language implemented on top of MapReduce.

The modular Java MapReduce style mirrors the logical structure found in the FlumeJava program, but it is not the normal way such computations would be expressed in MapReduce. The hand-optimized style represents an efficient execution strategy for the computation, and as such is much more common in practice than the modular version, but as a result of being hand-optimized and represented directly in terms of MapReduces, the logical computation can become obscured and hard to change. The hand-optimized Sawzall version likewise intermixes logical computation with lower-level implementation details, in an effort to get better performance.

The following table shows the number of lines of source it took to write each version of each benchmark:

Benchmark	FlumeJava	MapReduce (Modular)	MapReduce (Hand-Opt)	Sawzall (Hand-Opt)
Ads Logs	320	465	399	158
IndexStats	176	296	336	-
Build Logs	276	476	355	-
SiteData	465	653	625	261

For each case, the FlumeJava version is more concise than the equivalent version written using raw Java MapReduce. Sawzall is more concise than Java.

The following table presents, for the FlumeJava version, the number of FlumeJava operations in the pipeline (both before and after optimization), and for the Java MapReduce and Sawzall versions, the number of MapReduce stages:

| Benchmark	FlumeJava	MapReduce (Modular)	MapReduce (Hand-Opt)	Sawzall (Hand-Opt)
Ads Logs	$14 \rightarrow 1$	4	1	4
IndexStats	$16 \rightarrow 2$	3	2	-
Build Logs	$7 \rightarrow 1$	3	1	-
SiteData	$12 \rightarrow 2$	5	2	6

For each benchmark, the number of automatically optimized operations in the FlumeJava version matches the number of MapReduce stages in the corresponding hand-optimized MapReduce-based version. As a higher-level, domain-specific language, Sawzall does not provide the programmer sufficient low-level access to enable them to hand-optimize their programs into this minimum number of MapReduce stages, nor does it include an automatic optimizer.

The following table shows, for each benchmark, the size of the input data set and the number of worker machines we used to run it:

Benchmark	Input Size	Number of Machines
Ads Logs	550 MB	4
IndexStats	3.3 TB	200
Build Logs	34 GB	15
SiteData	1.3 TB	200

We compared the run-time performance of the different versions of each benchmark. We ensured that each version used equivalent numbers of machines and other resources. We measured the total elapsed wall-clock time spent when MapReduce workers were running; we excluded the "coordination" time of starting up the main controller program, distributing compiled binaries to worker machines, and cleaning up temporary files. Since execution times can vary significantly across runs, we ran each benchmark version five times, and took the minimum measured time as an approximation of the "true" time undisturbed by unrelated effects of running on a shared cluster of machines.

The chart below shows the elapsed time for each version of each benchmark, relative to the elapsed time for the FlumeJava version (shorter bars are better):
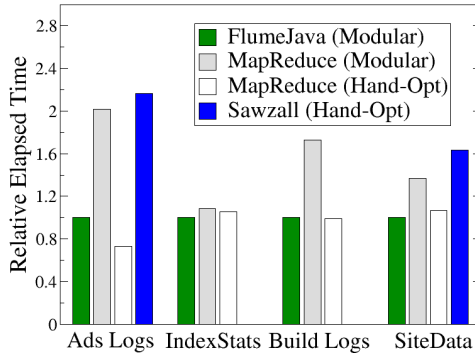
Comparing the two MapReduce columns and the Sawzall column shows the importance of optimizing. Without optimizations, the set-up overheads for the workers, the extra I/O in order to store the intermediate data, extra data encoding and decoding time, and other similar factors increase the overall work required to produce the output. Comparing the FlumeJava and the hand-optimized MapReduce columns demonstrates that a modular program written in FlumeJava runs at close to the performance of a hand-optimized version using the lower-level MapReduce APIs.

7. Related Work

In this section we briefly describe related work, and compare FlumeJava to that work.

Language and library support for data-parallel programming has a long history. Early work includes *Lisp [13], C* [18], C** [12], and pH [15].

MapReduce [6–8] combines simple abstractions for data-parallel processing with an efficient, highly scalable, fault-tolerant implementation. MapReduce's abstractions directly support computations that can be expressed as a map step, a shuffle step, and a reduce step. MapReduces can be programmed in several languages, including C++ and Java. FlumeJava builds on Java MapReduce, offering higher-level, more-composable abstractions, and an optimizer for recovering good performance from those abstractions. FlumeJava builds in support for managing pipelines of MapReduces. FlumeJava also offers additional conveniences that help make developing a FlumeJava program similar to developing a regular single-process Java program.

Sawzall [17] is a domain-specific logs-processing language that is implemented as a layer over MapReduce. A Sawzall program can flexibly specify the mapper part of a MapReduce, as long as the mappers are pure functions. Sawzall includes a library of a dozen standard reducers; users cannot specify their own reducers. This limits the Sawzall user's ability to express efficient execution plans for some computations, such as joins. Like MapReduce, Sawzall does not provide help for multi-stage pipelines.

Hadoop [2] is an open-source Java-based re-implementation of MapReduce, together with a job scheduler and distributed file system akin to the Google File System [9]. As such, Hadoop has similar limitations as MapReduce when developing multi-stage pipelines.

Cascading [1] is a Java library built on top of Hadoop. Like FlumeJava, Cascading aims to ease the challenge of programming data-parallel pipelines, and provides abstractions similar to those of FlumeJava. Unlike FlumeJava, a Cascading program explicitly constructs a dataflow graph. In addition, the values flowing through a Cascading pipeline are special untyped "tuple" values, and Cascading operations focus on transforms over tuples; in contrast, a FlumeJava pipeline computes over arbitrary Java objects using arbitrary Java computations. Cascading performs some optimizations of its dataflow graphs prior to running them. Somewhat akin to FlumeJava's executor, the Cascading evaluator breaks the dataflow graph into pieces, and, if possible, runs those in parallel, using the underlying Hadoop job scheduler. There is a mechanism for eliding computation if input data is unchanged, akin to FlumeJava's caching mechanism.

Pig [3] compiles a special domain-specific language called Pig Latin [16] into code that is run on Hadoop. A Pig Latin program combines high-level declarative operators similar to those in SQL, together with named intermediate variables representing edges in the dataflow graph between operators. The language allows for user-defined transformation and extraction functions, and provides support for co-grouping and joins. The Pig system has a novel debugging mechanism, wherein it can generate sample data sets that illustrate what the various operations do. The Pig system has an optimizer that tries to minimize the amount of data materialized between Hadoop jobs, and is sensitive to the size of the input data sets.

The Dryad [11] system implements a general-purpose data-parallel execution engine. Dryad programs are written in C++ using overloaded operators to specify an arbitrary acyclic dataflow graph, somewhat akin to Cascading's model of explicit graph construction. Like MapReduce, Dryad handles the details of communication, partitioning, placement, concurrency and fault tolerance. Unlike stock MapReduce but similar to the FlumeJava optimizer's MSCR primitive, computation nodes can have multiple input and output edge "channels." Unlike FlumeJava, Dryad does not have an optimizer to combine or rearrange nodes in the dataflow graph, since the nodes are computational black boxes, but Dryad does include a notion of run-time graph refinement through which users can perform some kinds of optimizations.

The LINQ [14] extension of C# 3.0 adds a SQL-like construct to C#. This construct is syntactic sugar for a series of library calls, which can be implemented differently over different kinds of data being "queried." The SQL-like construct can be used to express queries over traditional relational data (and shipped out to remote database servers), over XML data, and over in-memory C# objects. It can also be used to express parallel computations and executed on Dryad [20]. The manner in which the SQL-like construct is desugared into calls that construct an internal representation of the original query is similar to how FlumeJava's parallel operations

implicitly construct an internal execution plan. DryadLINQ also includes optimizations akin to those in FlumeJava. The C# language was significantly extended in order to support LINQ; in contrast, FlumeJava is implemented as a pure Java library, with no language changes. DryadLINQ requires a pipeline to be expressed via a single SQL-like statement. In contrast, calls to FlumeJava operations can be intermixed with other Java code, organized into functions, and managed with traditional Java control-flow operations; deferred evaluation enables all these calls to be coalesced dynamically into a single pipeline, which is then optimized and executed as a unit.

SCOPE [4] is a declarative scripting language built on top of Dryad. Programs are written in a variant of SQL, with extensions to call out to custom extractors, filters, and processors that are written in C#. The C# extensions are intermixed with the SQL code. As with Pig Latin, SQL queries are broken down into a series of distinct steps, with variables naming intermediate streams. The SQL framework provides named data fields, but there appears to be little support for those names in the extension code. The optimizer transforms SQL expressions using traditional rules for query optimization, together with new rules that take into account data and communication locality.

Map-Reduce-Merge [19] extends the MapReduce model by adding an additional Merge step, making it possible to express additional types of computations, such as relational algebra, in a single execution. FlumeJava supports more general pipelines.

FlumeJava's optimizer shares many concepts with traditional compiler optimizations, such as loop fusion and common-subexpression elimination. FlumeJava's optimizer also bears some resemblance to a database query optimizer: they both produce an optimized execution plan from a higher-level decription of a logical computation, and both can optimize programs that perform joins. However, a database query optimizer typically uses run-time information about input tables in a relational database, such as their sizes and available indices, to choose an efficient execution plan, such as which of several possible algorithms to use to compute joins. In contrast, FlumeJava provides no built-in support for joins. Instead, join() is a derived library operation that implements a particular join algorithm, hash-merge-join, which works even for simple, file-based data sets lacking indices and which can be implemented using MapReduce. Other join algorithms could be implemented by other derived library operations. FlumeJava's optimizer works at a lower level than a typical database query optimizer, applying fusion and other simple transformations to the primitives underlying the join() library operation. It chooses how to optimize without reference to the sizes or other representational properties of its inputs. Indeed, in the context of a join embedded in a large pipeline, such information may not become available until after the optimized pipeline has been partly run. FlumeJava's approach allows the operations implementing the join to be optimized in the context of the surrounding pipeline; in many cases the joining operations are completely fused into the rest of the computation (and vice versa). This was illustrated by the SiteData example in section 4.5.

Before FlumeJava, we were developing a system based on similar abstractions, but made available to users in the context of a new programming language, named Lumberjack. Lumberjack was designed to be particularly good for expressing data-parallel pipelines, and included features such as an implicitly parallel, mostly functional programming model, a sophisticated polymorphic type system, local type inference, lightweight tuples and records, and first-class anonymous functions. Lumberjack was supported by a powerful optimizer that included both traditional optimizations such as inlining and value flow analysis, and non-traditional optimizations such as fusion of parallel loops. Lumberjack programs were transformed into a low-level intermediate representation, which in our implementation was interpreted but which we planned to eventually dynamically translate into Java bytecode or native machine code. Lumberjack's parallel run-time system shared many of the characteristics of FlumeJava's run-time system.

While the Lumberjack-based version of Flume offered a number of benefits for programmers, it suffered from several important disadvantages relative to the FlumeJava version:

- Since Lumberjack was specially designed for the task, Lumberjack programs were significantly more concise than the equivalent FlumeJava programs. However, the implicitly parallel, mostly functional programming model was not natural for many of its intended users. FlumeJava's explicitly parallel model, which distinguishes `Collection` from `PCollection` and `iterator()` from `parallelDo()`, coupled with its "mostly imperative" model that disallows mutable shared state only across `DoFn` boundaries, is much more natural for most of these programmers.

- Lumberjack's optimizer was a traditional static optimizer, which performed its optimization over the program's internal representation before executing any of it. Since FlumeJava is a pure library, it cannot use a traditional static optimization approach. Instead, we adopted a more dynamic approach to optimization, where the running user program first constructs an execution plan (via deferred evaluation), and then optimizes the plan before executing it. FlumeJava does no static analysis of the source program nor dynamic code generation, which imposes some costs in run-time performance; those costs have turned out to be relatively modest. On the other hand, being able to simply run the FlumeJava program to construct the fully expanded execution plan has turned out to be a tremendous advantage. The ability of Lumberjack's optimizer to deduce the program's execution plan was always limited by the strength of its static analysis, but FlumeJava's dynamic optimizer has no such limits. Indeed, FlumeJava programmers routinely use complex control structures and `Collections` and `Maps` storing `PCollections` in their code expressing their pipeline computation. These coding patterns would defeat any static analysis we could reasonably develop, but the FlumeJava dynamic optimizer is unaffected by this complexity. Later, we can augment FlumeJava with a dynamic code generator, if we wish the reduce the remaining overheads.

- Building an efficient, complete, usable Lumberjack-based system is much more difficult and time-consuming than building an equivalently efficient, complete, and usable FlumeJava system. Indeed, we had built only a prototype Lumberjack-based system after more than a year's effort, but we were able to change directions and build a useful FlumeJava system in only a couple of months.

- Novelty is an obstacle to adoption. By being embedded in a well-known programming language, FlumeJava focuses the potential adopter's attention on a few new features, namely the Flume abstractions and the handful of Java classes and methods implementing them. Potential adopters are not distracted by a new syntax or a new type system or a new evaluation model. Their normal development tools and practices continue to work. All the standard libraries they have learned and rely on are still available. They need not fear that Java will go away and leave their project in the lurch. By comparison, Lumberjack suffered greatly along these dimensions. The advantages of its specially designed syntax and type system were insufficient to overcome these real-world obstacles.

8. Conclusion

FlumeJava is a pure Java library that provides a few simple abstractions for programming data-parallel computations. These abstractions are higher-level than those provided by MapReduce, and provide better support for pipelines. FlumeJava's internal use of a form of deferred evaluation enables the pipeline to be optimized prior to execution, achieving performance close to that of hand-optimized MapReduces. FlumeJava's run-time executor can select among alternative implementation strategies, allowing the same program to execute completely locally when run on small test inputs and using many parallel machines when run on large inputs. FlumeJava is in active, production use at Google. Its adoption has been facilitated by being a "mere" library in the context of an existing, well-known, expressive language.

References

[1] Cascading. `http://www.cascading.org`.

[2] Hadoop. `http://hadoop.apache.org`.

[3] Pig. http://hadoop.apache.org/pig.

[4] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2), 2008.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] J. Dean. Experiences with MapReduce, an abstraction for large-scale computation. In *Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51, no. 1, 2008.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[10] R. H. Halstead Jr. New ideas in parallel Lisp: Language design, implementation, and programming tools. In *Workshop on Parallel Lisp*, 1989.

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

[12] J. R. Larus. C**: A large-grain, object-oriented, data-parallel programming language. In *Languages and Compilers for Parallel Computing (LCPC)*, 1992.

[13] C. Lasser and S. M. Omohundro. The essential Star-lisp manual. Technical Report 86.15, Thinking Machines, Inc., 1986.

[14] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling objects, relations and XML in the .NET framework. In *ACM SIGMOD International Conference on Management of Data*, 2006.

[15] R. S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Academic Press, 2001.

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

[17] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

[18] J. R. Rose and G. L. Steele Jr. C*: An extended C language. In *C++ Workshop*, 1987.

[19] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *ACM SIGMOD International Conference on Management of Data*, 2007.

[20] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.