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

FlumeJava: Easy, Efficient Data-Parallel Pipelines

Miroslav Cupák

11/04/2013

Outline

- Introduction
- MapReduce
- Abstractions & Deferred Evaluation
- Optimizer & Executor
- Evaluation
- Drawbacks and Future Work
- Related Work
- Conclusion

Introduction

- basic premise
  - MapReduce is very successful
  - it’s hard to write “raw” MapReduce programs
  - let's treat MapReduce as an underlying execution engine for a higher-level "language"
- goal
  - make it easy to develop, test and run efficient data-parallel pipelines

MapReduce

- programming model for processing large data
- key/value pairs
- functions: map, reduce (partitioning, combiner)
- custom input/output types
- automatic parallelization and execution
- data automatically partitioned and machine failures handled using reexecution
- takes advantage of locality and backup tasks

MapReduce

- 3 phases
  - map
    - reads K/V pairs from the storage
    - applies Map to each element in parallel
  - shuffle
    - takes K/V pairs from Maps and groups them
  - reduce
    - applies Reducer to each distinct K/V group in parallel
    - aggregates the data and writes to a storage

MapReduce

- does
  - low-level selection of workers
  - distributes the program
  - manages temporary storage and data flow
  - handles failures
- does NOT do
  - many computations require a sequence/graph of MRs
  - high-level operations (e.g. joins) must be hand-rewritten to low-level MRs
  - explicit data flow and cleanup of intermediate files

FlumeJava

- Java library
- a few simple high-level abstractions for data-parallel computations and their pipelining
- focused on a logical view of the computation
- deferred evaluation and optimization of execution plan
- dynamic choice of local/distributed execution
- widely used at Google

Core Abstractions

- parallel collections
  - *PCollection<T>*
  - *PTable<K, V>*
- parallel operations
  - *parallelDo()*
  - *groupByKey()*
  - *combineValues()*
  - *flatten()*
  - derived operations: *count()*, *join()*, *top()*

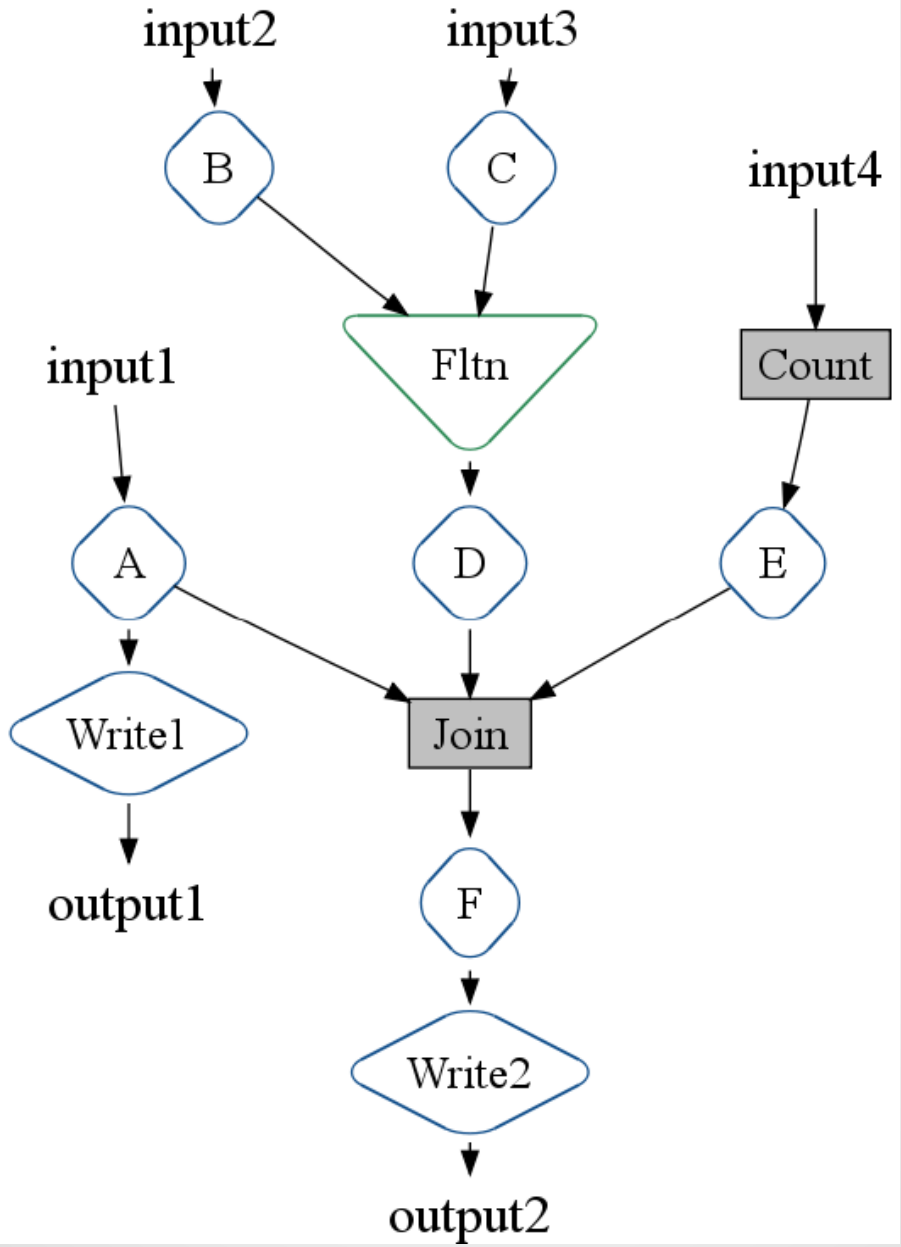
Core Abstractions - Example

```
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);
```

Deferred Evaluation

- lazy execution of operations
- 2 *PCollection* states: deferred and materialized
- *PCollection* holds a pointer for its operation
- operation references its *PCollection* arguments and results
- *parallelDo()* creates an object and returns a deferred *PCollection* pointing to it
- execution plan: directed acyclic graph

Deferred Evaluation - Example



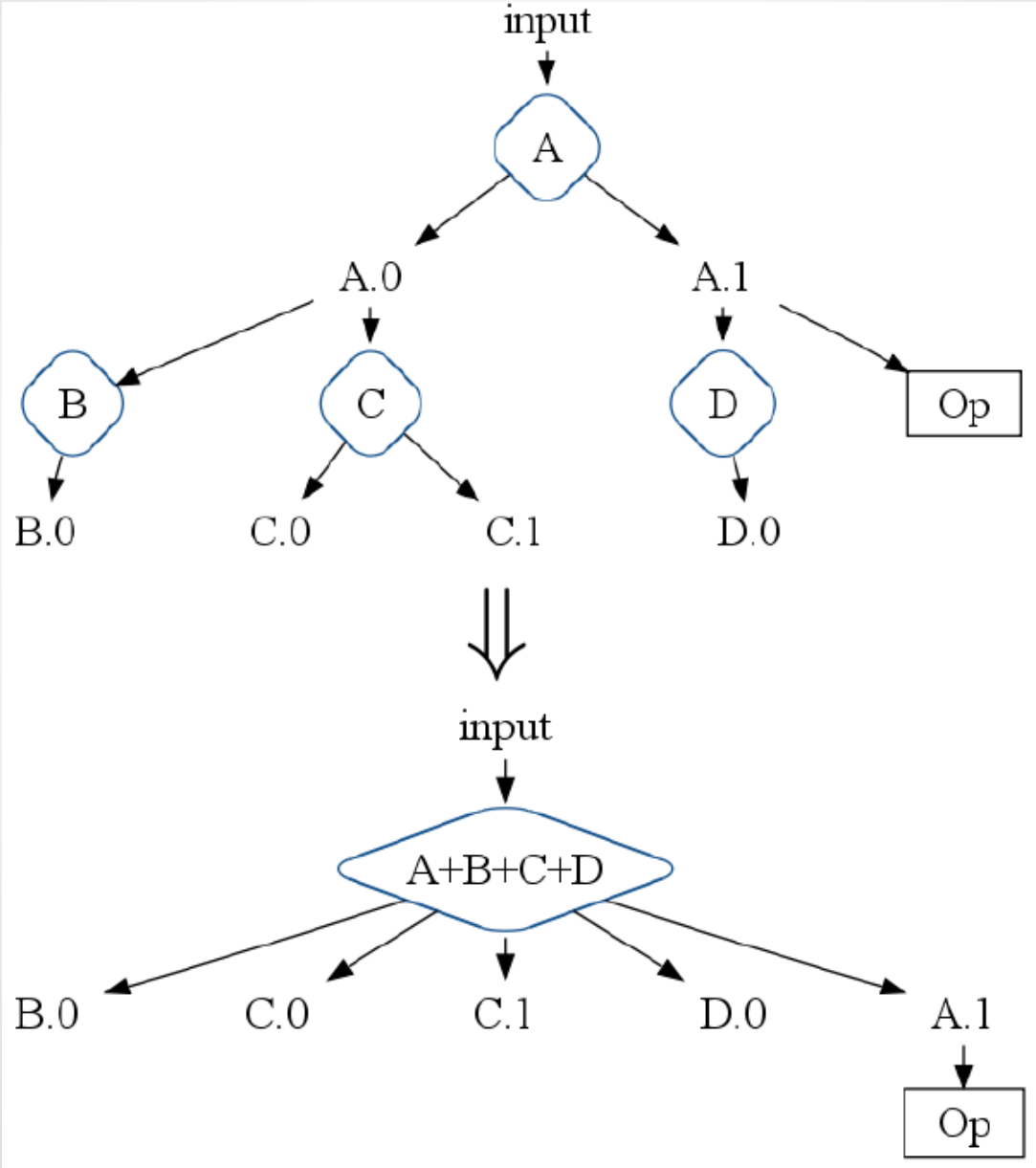
Deferred Evaluation

- actual evaluation with *FlumeJava.run()*
- *PObject<T>*
  - container for a typed object
  - used for results of deferred operations
  - similar to a future
  - value accessed after/during execution of a pipeline via *getValue()/operate()*

Optimizer

- transforms user-constructed modular execution plan into an efficient plan
- essentially a series of graph transformations
- basic operation: ParallelDo fusion
  - producer-consumer fusion
  - sibling fusion

ParallelDo Fusion - Example



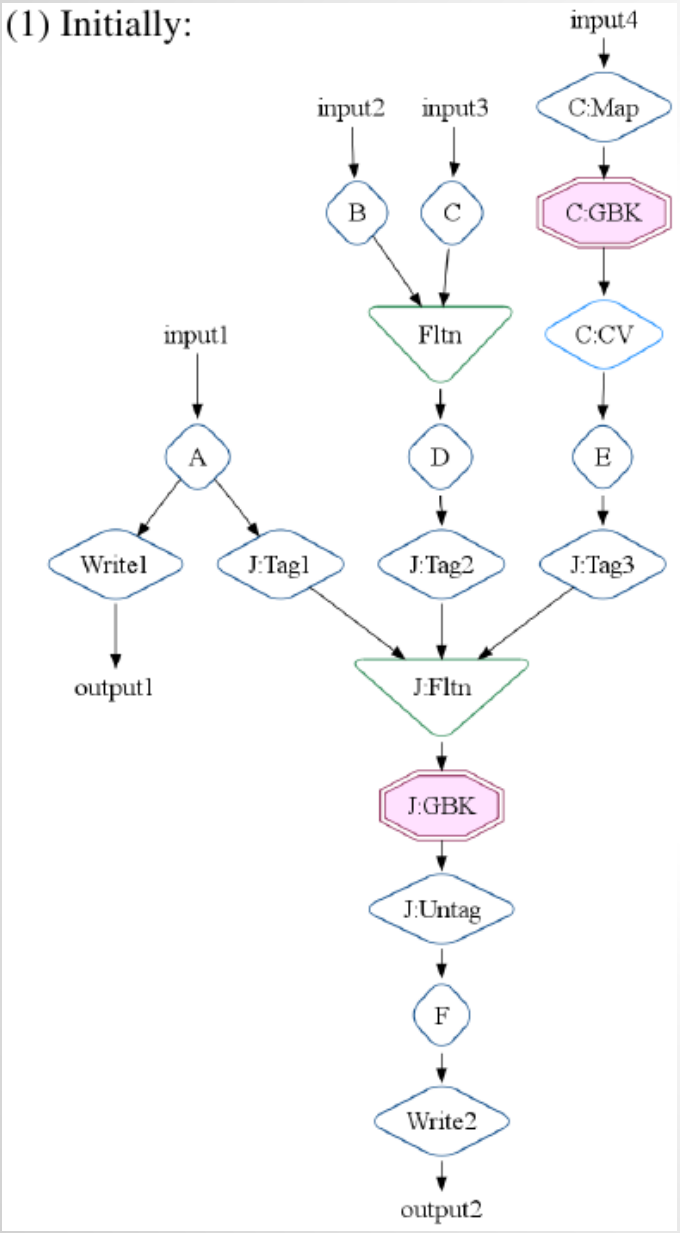
Optimizer

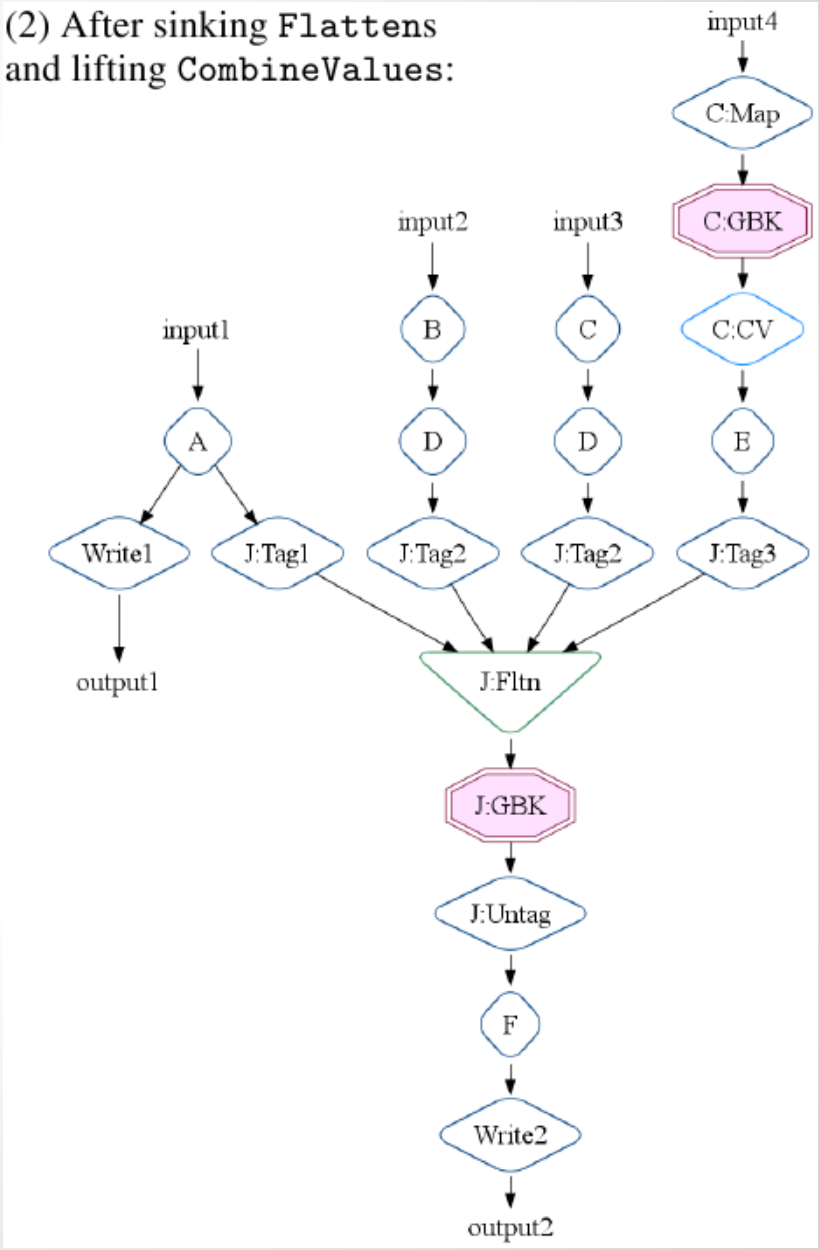
- core functionality
  - transformation of ParallelDo, GroupByKey, CombineValues, Flatten operations into single MRs
- MSCR
  - MapShuffleCombineReduce operation
  - intermediate step in the transformation
  - structure:
    - M input channels (maps)
    - R output channels (reduces) with shuffle/combine

Optimizer

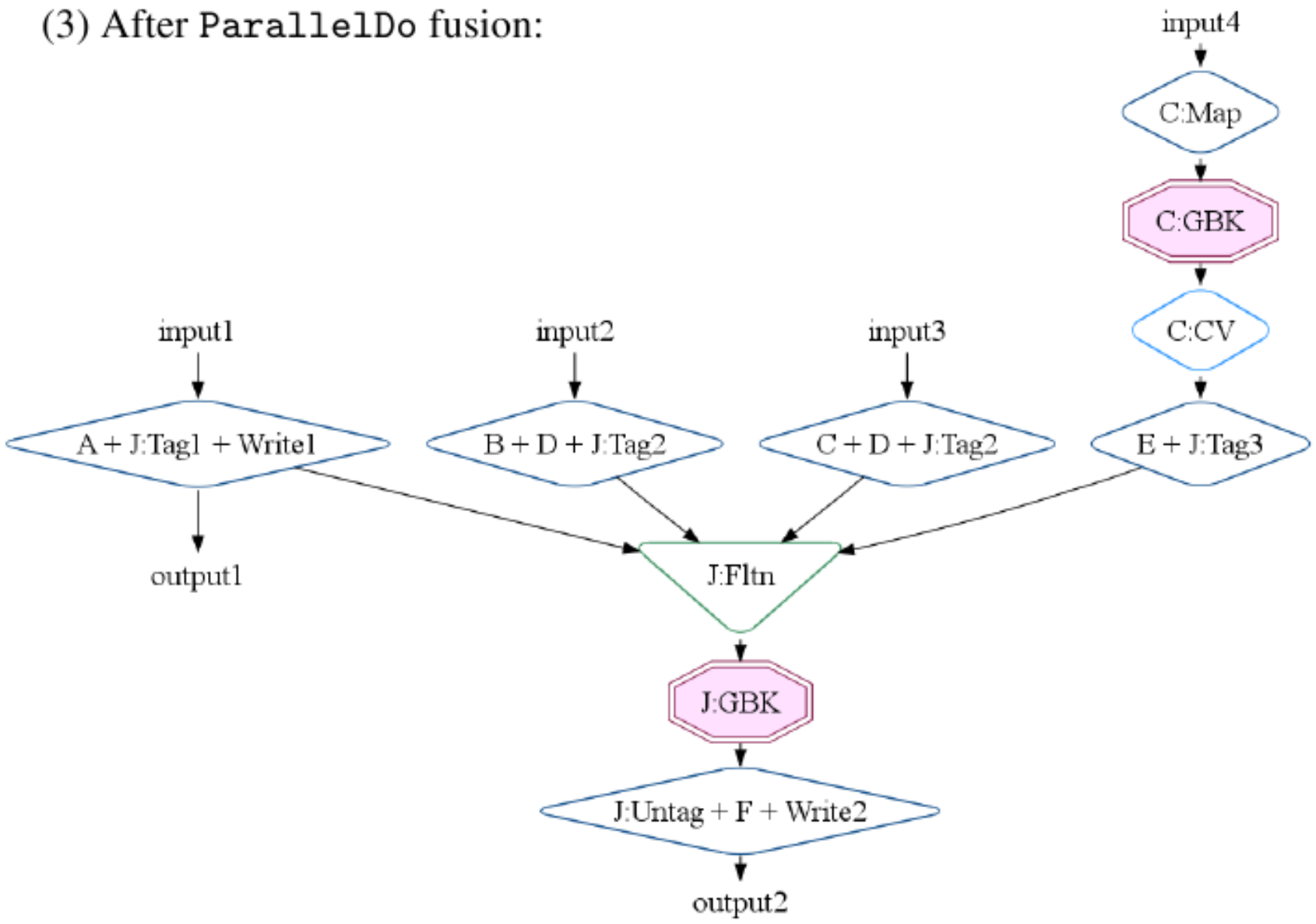
- MSCR fusion
  - each GroupByKey starts an output channel
  - ParallelDo forms an input channel
  - other GroupByKey inputs form input channels with identity mappers
  - CombineValues fuse into output channels
  - following ParallelDos are fused into output channel if it doesn't fit into the next input channel
  - unnecessary collections and CombineValues removed
  - ParallelDos with other outputs create a channel

Optimizer - Strategy

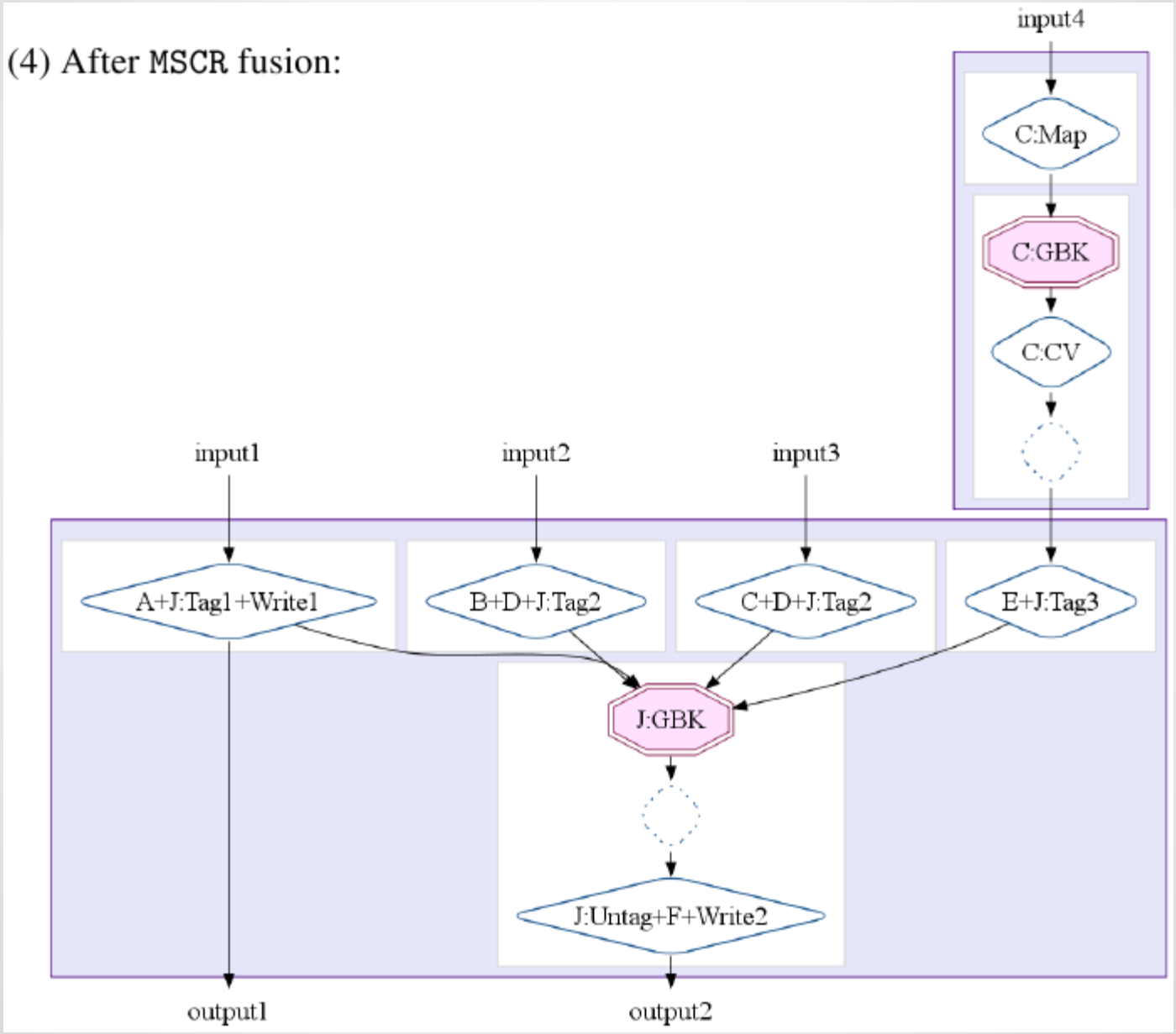




Optimizer - Strategy



Optimizer - Strategy

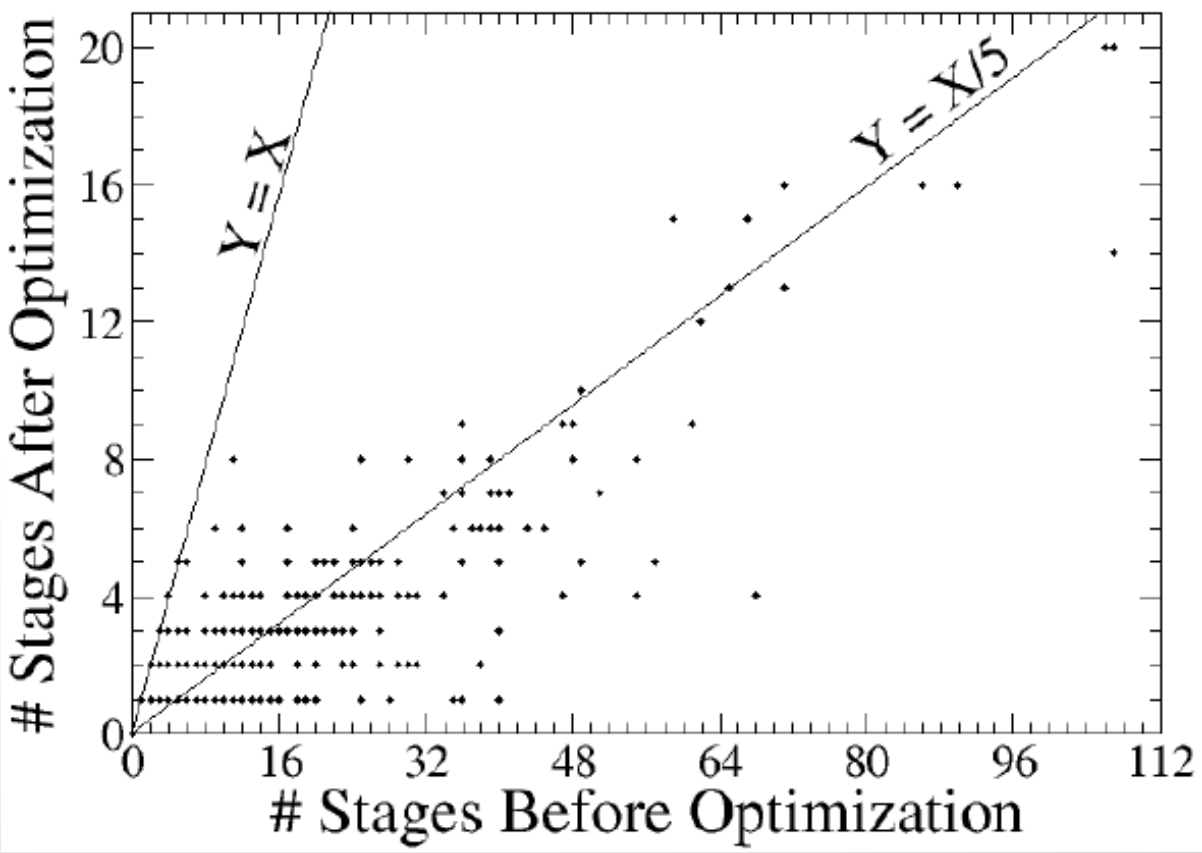


Executor

- batch execution
- traverses and executes the operations
- independent operations executed in parallel
- executes MSCR locally/remotely in parallel based on the data set (or user hint)
- automatically deletes temporary files
- caches results to support debugging

Evaluation

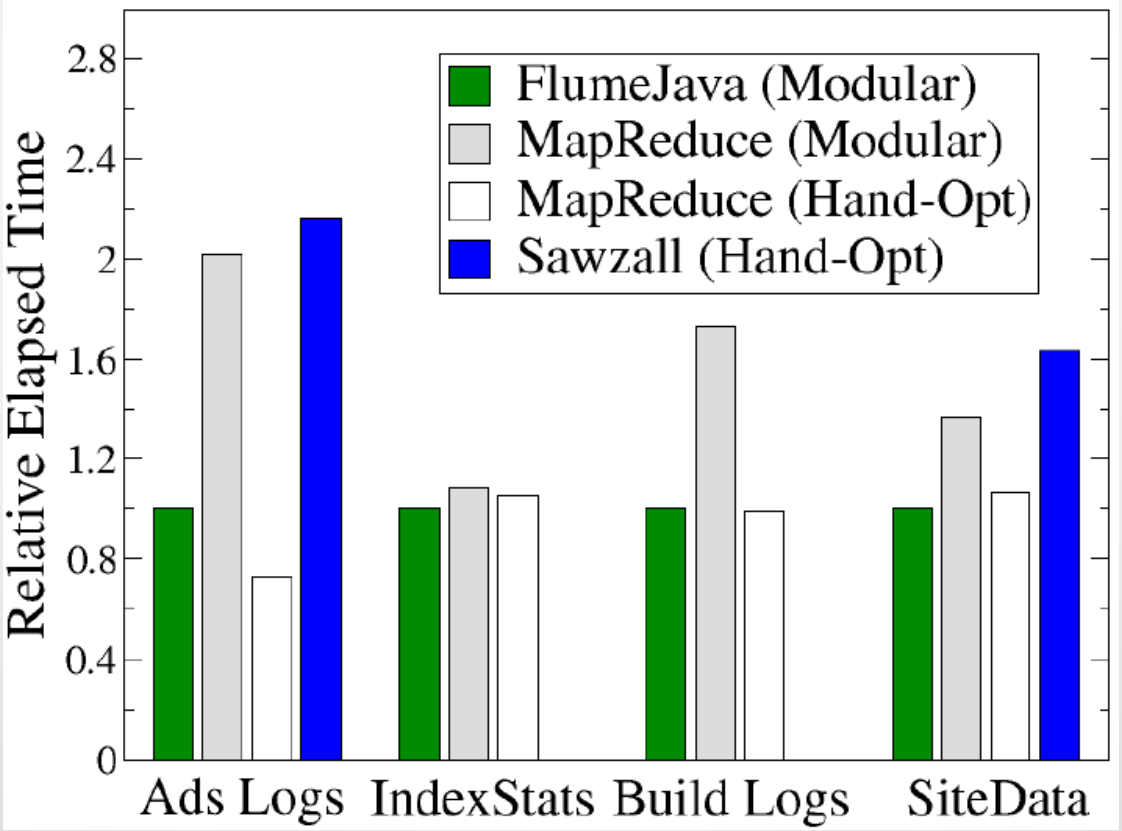
- user adoption and experience
  - 319 users, x*100 apps (May 2009 - March 2010)
- optimizer effectiveness



Evaluation

- execution performance

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



Current Drawbacks and Future Work

- expressiveness of low-level primitives
- use of static analysis might improve things
- optimizations do not involve user code
- preserves unnecessary GroupByKeys
- no support for incremental or streaming execution of pipelines

Related Work

- MapReduce
- Sawzall
- Hadoop+Cascading
- Dryad+DryadLINQ
- Lumberjack

Conclusions

- FlumeJava
  - is an expressive Java library based on composable primitives
  - provides abstractions for programming of data-parallel computations
  - addresses limitations of MapReduce
  - creates effective execution plans
  - is successfully used

Thank you! Questions?