Data Engineering

LTAT.02.007

Ass Prof. Riccardo Tommasini

Assistants: Fabiano Spiga, Mohamed Ragab, Hassan Eldeeb

https://courses.cs.ut.ee/2020/dataeng

Forum

Moodle

FlumeJava[1]

Easy, Efficient Data-Parallel Pipelines

[1] The reason as to why Airflow does not support streaming is that there are no obvious behavior rules that could be set so that the airflow scheduler could deterministically check if it has been completed or not.

Problem

Problem

- MapReduce works well for computations that can be broken down into a map step, a shuffle step, and a reduce step

Problem

- MapReduce works well for computations that can be broken down into a map step, a shuffle step, and a reduce step
- Many real-world computations require a chain of MapReduce works, e.g., graph analytics

Problem

- MapReduce works well for computations that can be broken down into a map step, a shuffle step, and a reduce step
- Many real-world computations require a chain of MapReduce works, e.g., graph analytics
- Additional coordination code chains together the separate MapReduce stages, requiring additional work to manage the intermediate results

Solution

Solution

- FlumeJava is a new system that aims to support the development of data-parallel pipelines.

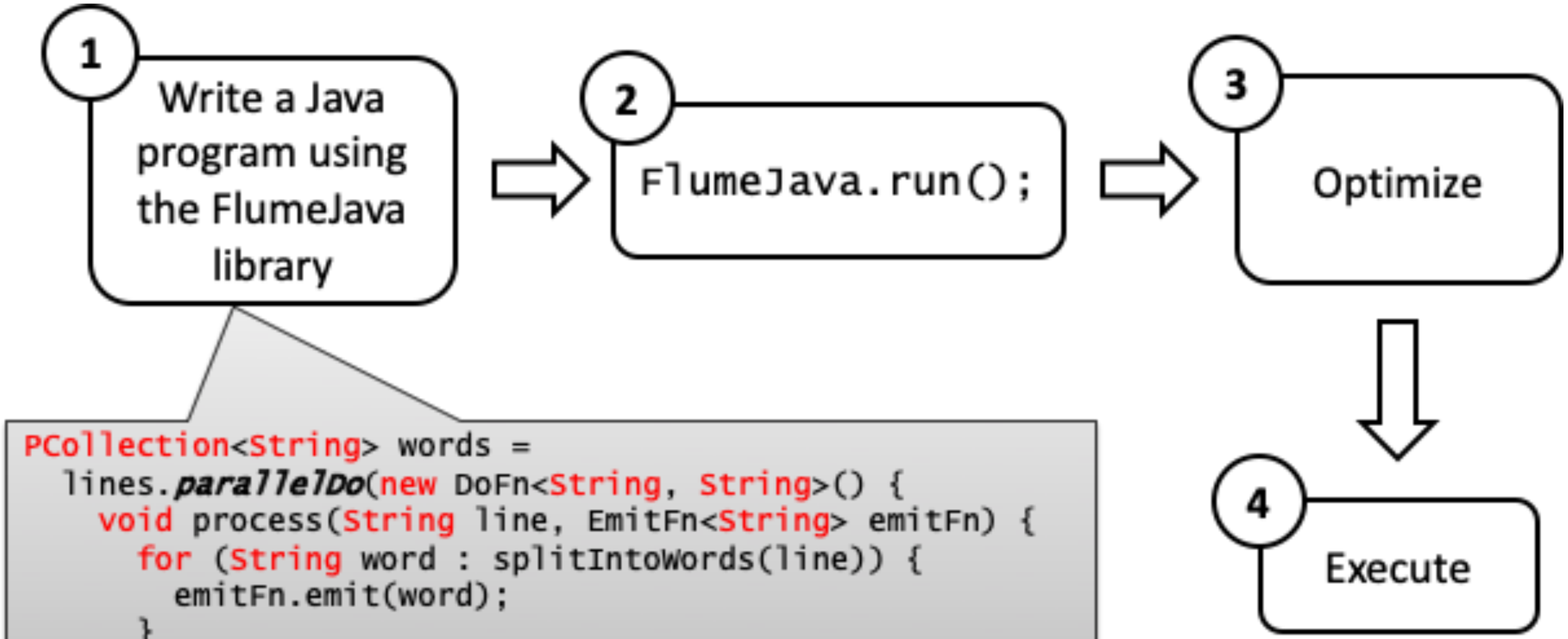
Solution

- FlumeJava is a new system that aims to support the development of data-parallel pipelines.
- Abstract away the details of how data is represented, including whether the data is represented as an in-memory data structure

Solution

- FlumeJava is a new system that aims to support the development of data-parallel pipelines.
- Abstract away the details of how data is represented, including whether the data is represented as an in-memory data structure
- FlumeJava evaluation is lazy: 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.

FlumeJava Workflow



Core Abstractions

Parallel Collections

Parallel Collections

- **PCollection**<T> a (possibly huge) immutable bag of elements of type T.

Parallel Collections

- **PCollection**<T> a (possibly huge) immutable bag of elements of type T.
  - If it has a well-defined order is called a Sequence

Parallel Collections

- **PCollection**<T> a (possibly huge) immutable bag of elements of type T.
  - If it has a well-defined order is called a Sequence
- **PTable**<K, V> represents a (possibly huge) immutable multi-map with keys of type K and values of type V.

Parallel Collections

- **PCollection**<T> a (possibly huge) immutable bag of elements of type T.
  - If it has a well-defined order is called a Sequence
- **PTable**<K, V> represents a (possibly huge) immutable multi-map with keys of type K and values of type V.
  - In Java, PTable<K,V> is a subclass of PCollection<Pair<K,V>>

Parallel Collections

- **PCollection**<T> a (possibly huge) immutable bag of elements of type T.
  - If it has a well-defined order is called a Sequence
- **PTable**<K, V> represents a (possibly huge) immutable multi-map with keys of type K and values of type V.
  - In Java, PTable<K,V> is a subclass of PCollection<Pair<K,V>>
- **PObjects**<T> is a container for a single Java object of type T that acts much like a future.

Parallel Collections

- **PCollection**<T> a (possibly huge) immutable bag of elements of type T.
  - If it has a well-defined order is called a Sequence
- **PTable**<K, V> represents a (possibly huge) immutable multi-map with keys of type K and values of type V.
  - In Java, PTable<K,V> is a subclass of PCollection<Pair<K,V>>
- **PObjects**<T> is a container for a single Java object of type T that acts much like a future.
  - Like PCollections, PObjects can be either deferred or materialized.

Parallel Operations (Primitives)

Parallel Operations (Primitives)

- **parallelDo**() higher-order function that supports element-wise computation over an input PCollection<T>

Parallel Operations (Primitives)

- **parallelDo**() higher-order function that supports element-wise computation over an input PCollection<T>
- **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\>> [2]

---

[2] captures the essence of the shuffle step of MapReduce.

Parallel Operations (Primitives)

- **parallelDo**() higher-order function that supports element-wise computation over an input PCollection<T>
- **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\>> [2]
- **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[3]

---

[2] captures the essence of the shuffle step of MapReduce.

[3] combineValues() is a special case of parallelDo(). The associativity of the combining function allows it to be implemented via a combination of a MapReduce combiner and a MapReduce reducer.

Parallel Operations (Primitives)

- **parallelDo**() higher-order function that supports element-wise computation over an input PCollection<T>
- **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\>> [2]
- **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[3]
- **flatten**() takes a list of PCollection<T>s and returns a single PCollection<T>

---

[2] captures the essence of the shuffle step of MapReduce.

[3] combineValues() is a special case of parallelDo(). The associativity of the combining function allows it to be implemented via a combination of a MapReduce combiner and a MapReduce reducer.

WordCount in Primitive FlumeJava

Sentences to Words

```
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()));
```

Words Occurrences

```
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()));
```

Occurrences to Counts

```
PTable<String,Collection<Integer>> gwOnes = wordsWithOnes.groupByKey();

PTable<String,Integer> wordCounts = gwOnes.combineValues(SUM_INTS);
```

Derived operations

Derived operations

- **count**() 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. f parallelDo(), groupByKey(), and combineValues()

```
PTable<String,Collection<Integer>> gwOnes = wordsWithOnes.groupByKey();

PTable<String,Integer> wordCounts = gwOnes.combineValues(SUM_INTS);
```

OR

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

Derived operations

Derived operations

- **count**() 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. f parallelDo(), groupByKey(), and combineValues()

Derived operations

- **count**() 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. f parallelDo(), groupByKey(), and combineValues()
- **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 multimap PTable<K, V2>, join() returns a uni-map PTable<K,Tuple2<Collection<V1>, Collection<V2\>>>

Join

Join

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

Join

1. Apply **parallelDo**() to each input `PTable<K, Vi>` to convert it into a type `PTable<K,TaggedUnion2<V1,V2>>`.
2. Combine the tables using **flatten**().

Join

1. Apply **parallelDo**() to each input `PTable<K, Vi>` to convert it into a 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>>>`.

Join

1. Apply **parallelDo**() to each input `PTable<K, Vi>` to convert it into a 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<Collection<V1>,Collection<V2>>`.

Derived operations

Derived operations

- **count**() 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. f parallelDo(), groupByKey(), and combineValues()

Derived operations

- **count**() 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. f parallelDo(), groupByKey(), and combineValues()
- **join**() takes two or more `PTables` sharing a common key type. When applied to a multi-maps, it returns a uni-map `PTable<K,Tuple2<Collection<V1>,Collection<V2>>>`

Derived operations

- **count**() 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. f parallelDo(), groupByKey(), and combineValues()
- **join**() takes two or more `PTables` sharing a common key type. When applied to a multi-maps, it returns a uni-map `PTable<K,Tuple2<Collection<V1>,Collection<V2>>>`
- **top**() which takes a PCollection, a comparison function and returns the greatest N elements according to the comparison function.

Optimizer

Optimizer

- The interesting part of FlumeJava is its optimizer, which can transform a user-constructed FlumeJava program into one that can be executed efficiently.

Optimizer

- The interesting part of FlumeJava is its optimizer, which can transform a user-constructed FlumeJava program into one that can be executed efficiently.
- The optimizer is written as a series of independent graph transformations.

Optimizer

- The interesting part of FlumeJava is its optimizer, which can transform a user-constructed FlumeJava program into one that can be executed efficiently.
- The optimizer is written as a series of independent graph transformations.
- The optimizers include some intermediate-level operations (not exposed to the users) that combine FlumeJava primitives into Map Reduces jobs

Overall Optimizer Strategy

Overall Optimizer Strategy

- **Sink Flattens**

Overall Optimizer Strategy

- **Sink Flattens**
- **Lift CombineValues**

Overall Optimizer Strategy

- **Sink Flattens**
- **Lift CombineValues**
- **Insert fusion blocks**

Overall Optimizer Strategy

- **Sink Flattens**
- **Lift CombineValues**
- **Insert fusion blocks**
- **Fuse ParallelDos**

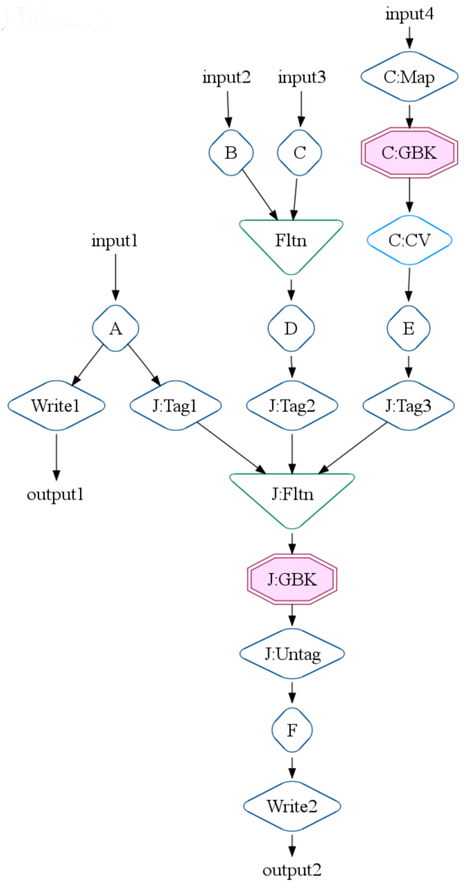
Overall Optimizer Strategy

- **Sink Flattens**
- **Lift CombineValues**
- **Insert fusion blocks**
- **Fuse ParallelDos**
- **MapShuffleCombineReduce (MSCR)**

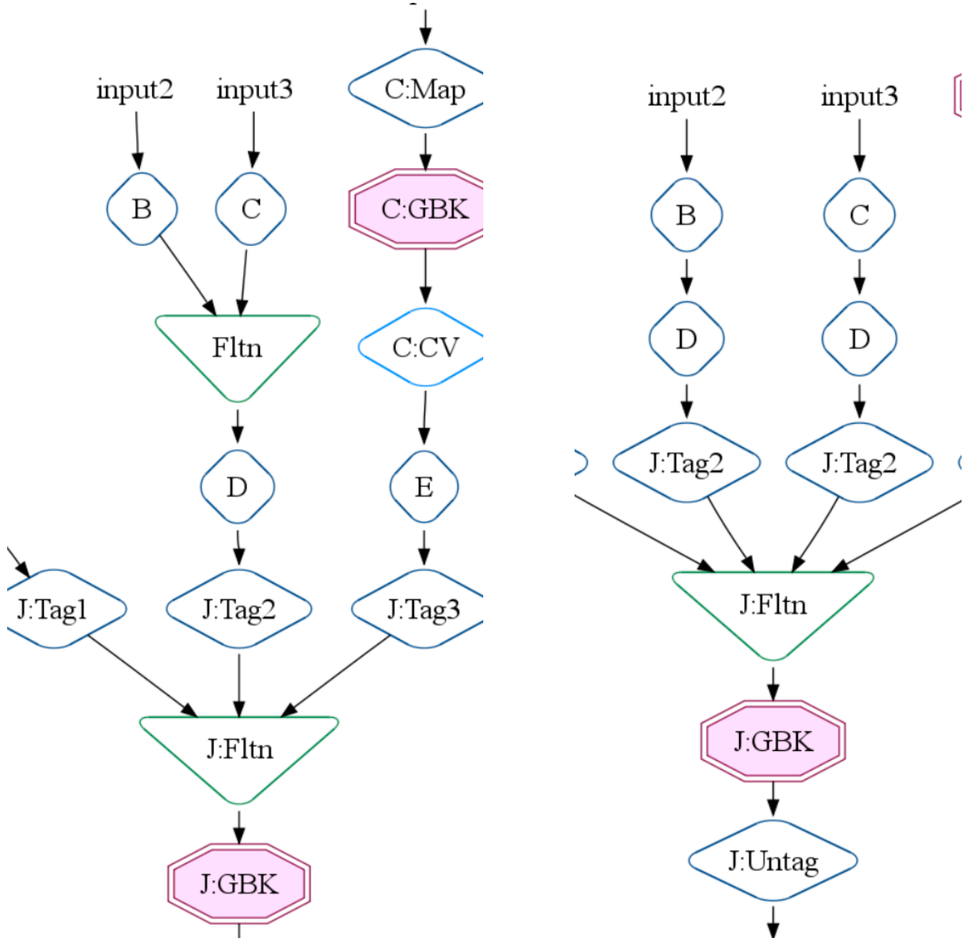
Overall Optimizer Strategy

- **Sink Flattens**
- **Lift CombineValues**
- **Insert fusion blocks**
- **Fuse ParallelDos**
- **MapShuffleCombineReduce (MSCR)**
- **Fuse MSCRs**

Running Example

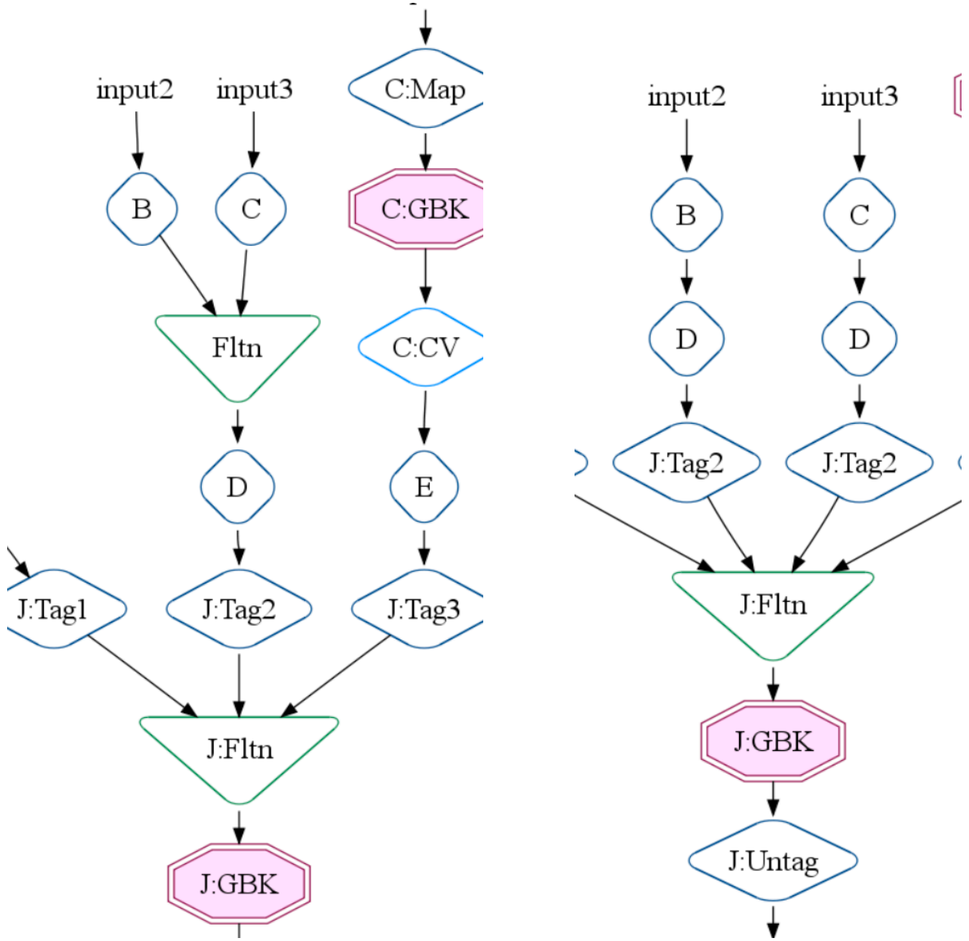


Sink Flattens.



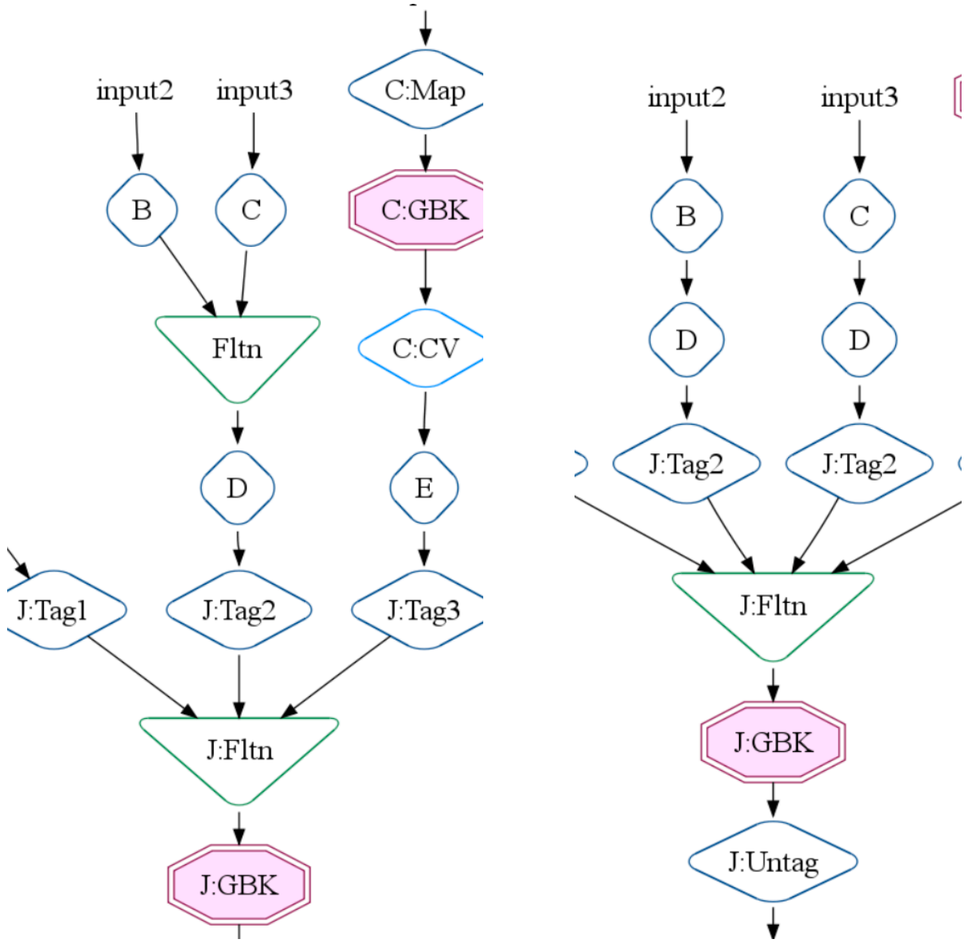
Sink Flattens.

- A Flatten operation can be pushed down through consuming ParallelDo operations by duplicating the ParallelDo before each input to the Flatten.



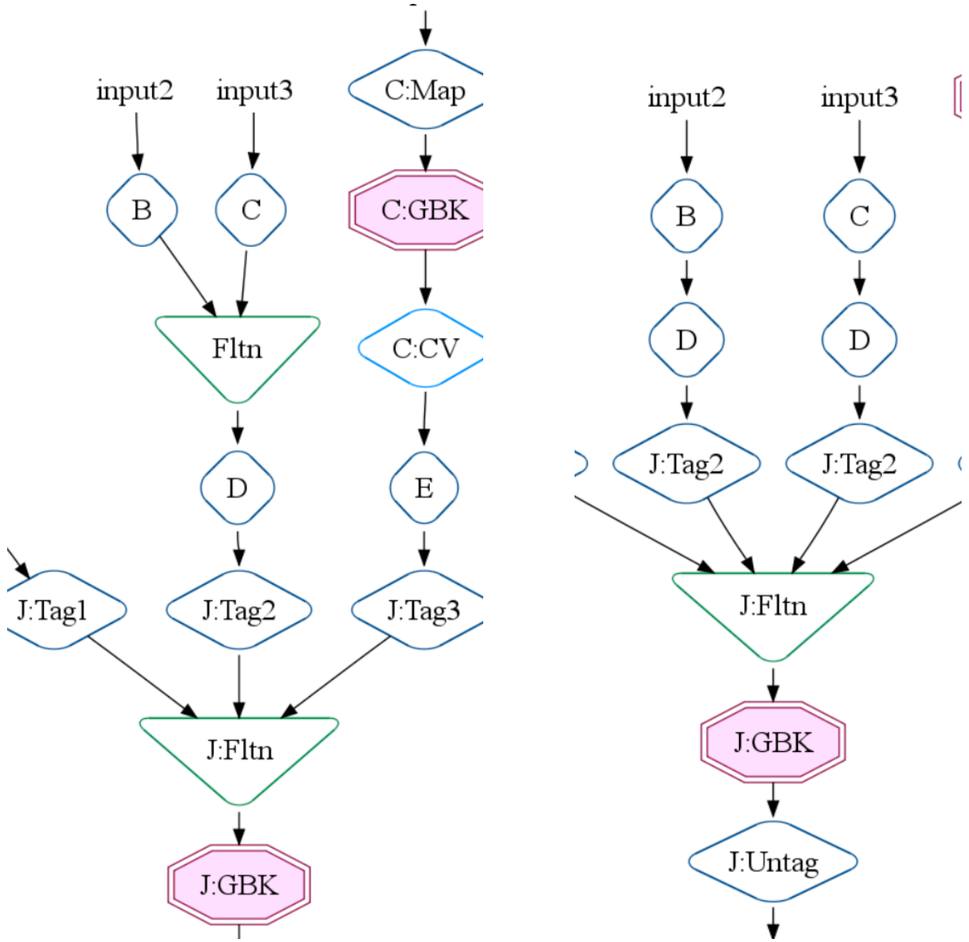
Sink Flattens.

- 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))`.

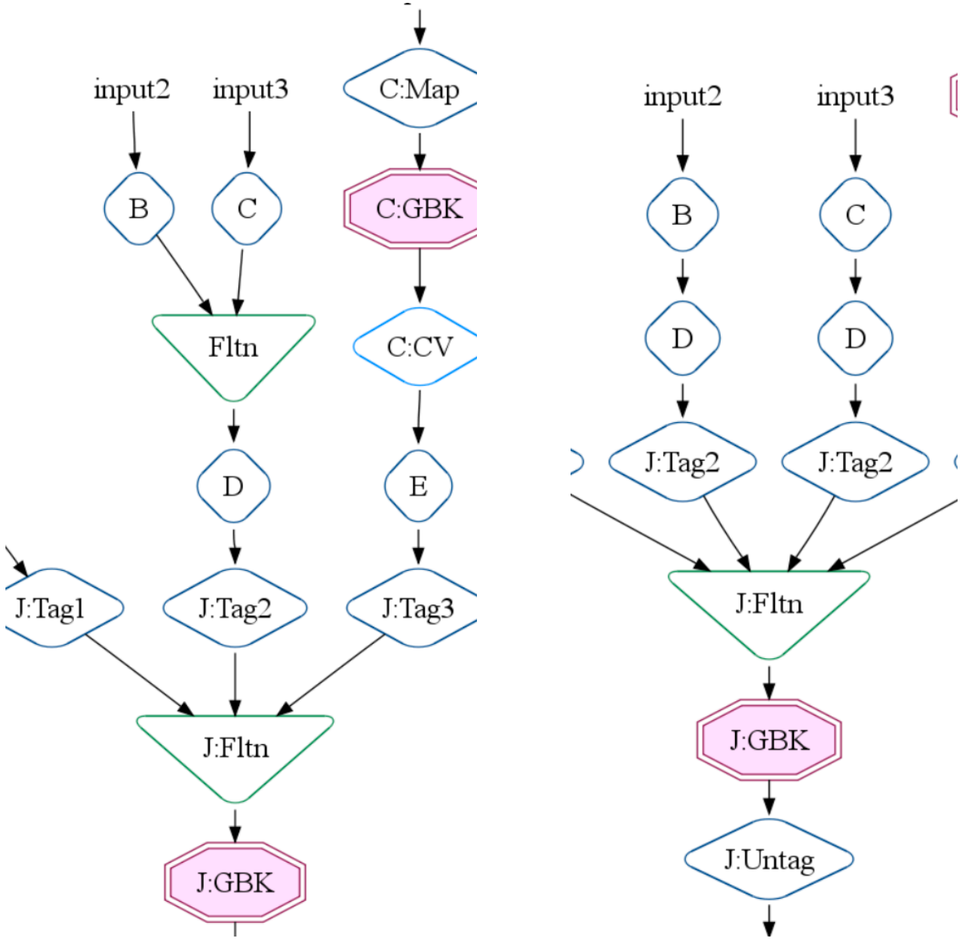


Sink Flattens.

- 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...

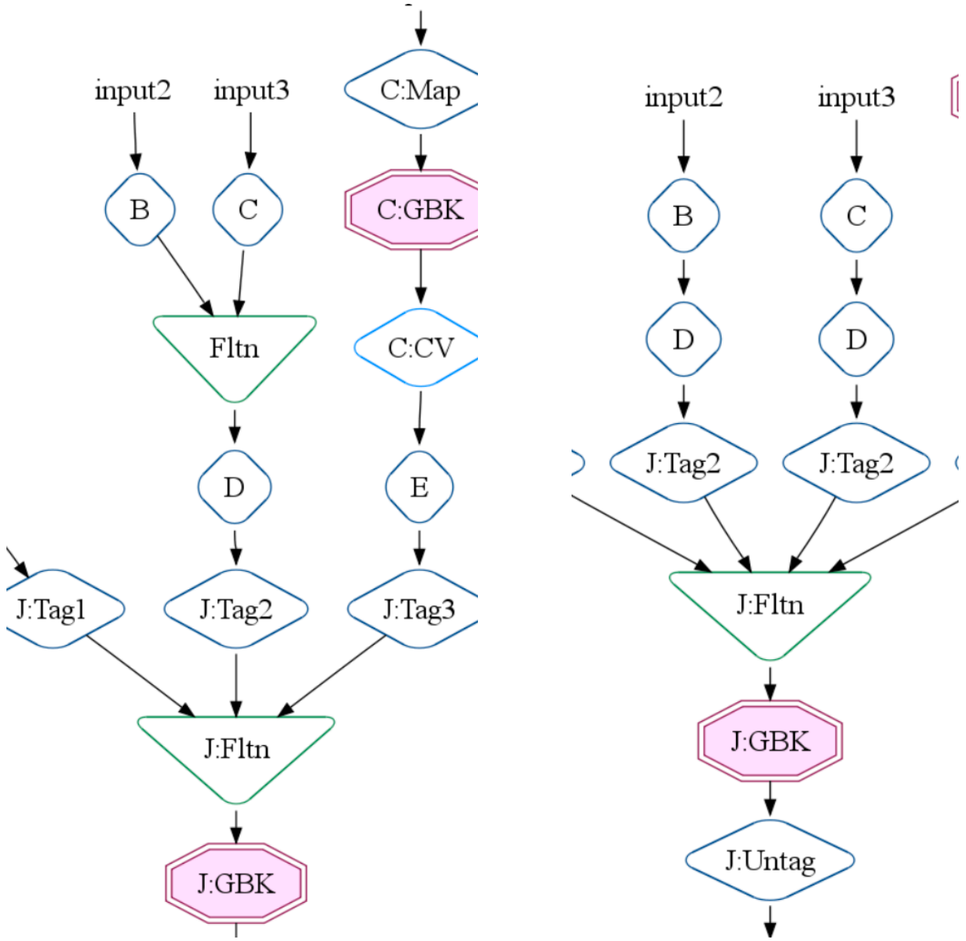


Lift CombineValues



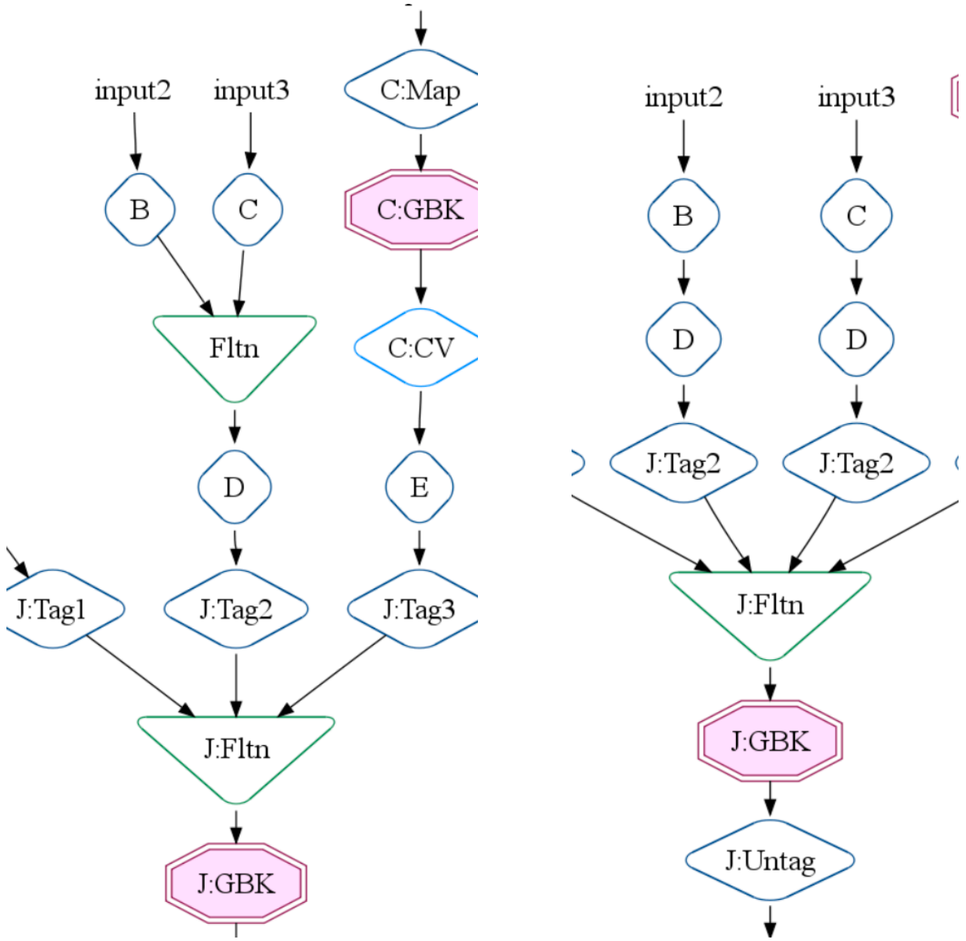
Lift CombineValues

- If a CombineValues operation immediately follows a GroupByKey operation, the CombineValues is henceforth treated as a normal ParallelDo operation and subject to ParallelDo fusion.



Lift CombineValues

- If a CombineValues operation immediately follows a GroupByKey operation, the CombineValues is henceforth treated as a normal ParallelDo operation and subject to ParallelDo fusion.
- The CombineValues operation C:CV is left in place and associated with C:GBK.



ParallelDo Fusion: function composition

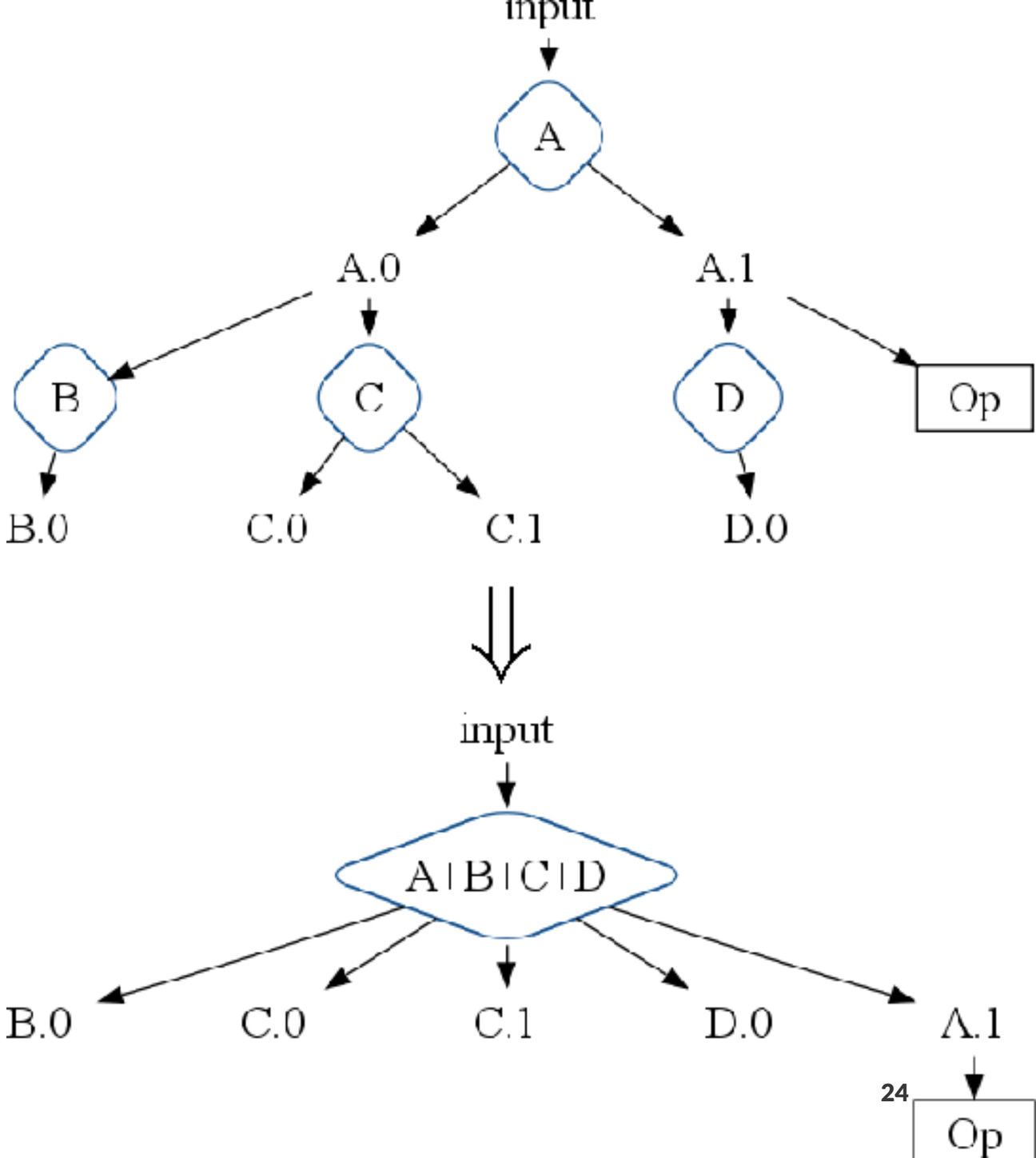
when: one ParallelDo operation performs function f, and its result is consumed by another ParallelDo operation that performs function g.

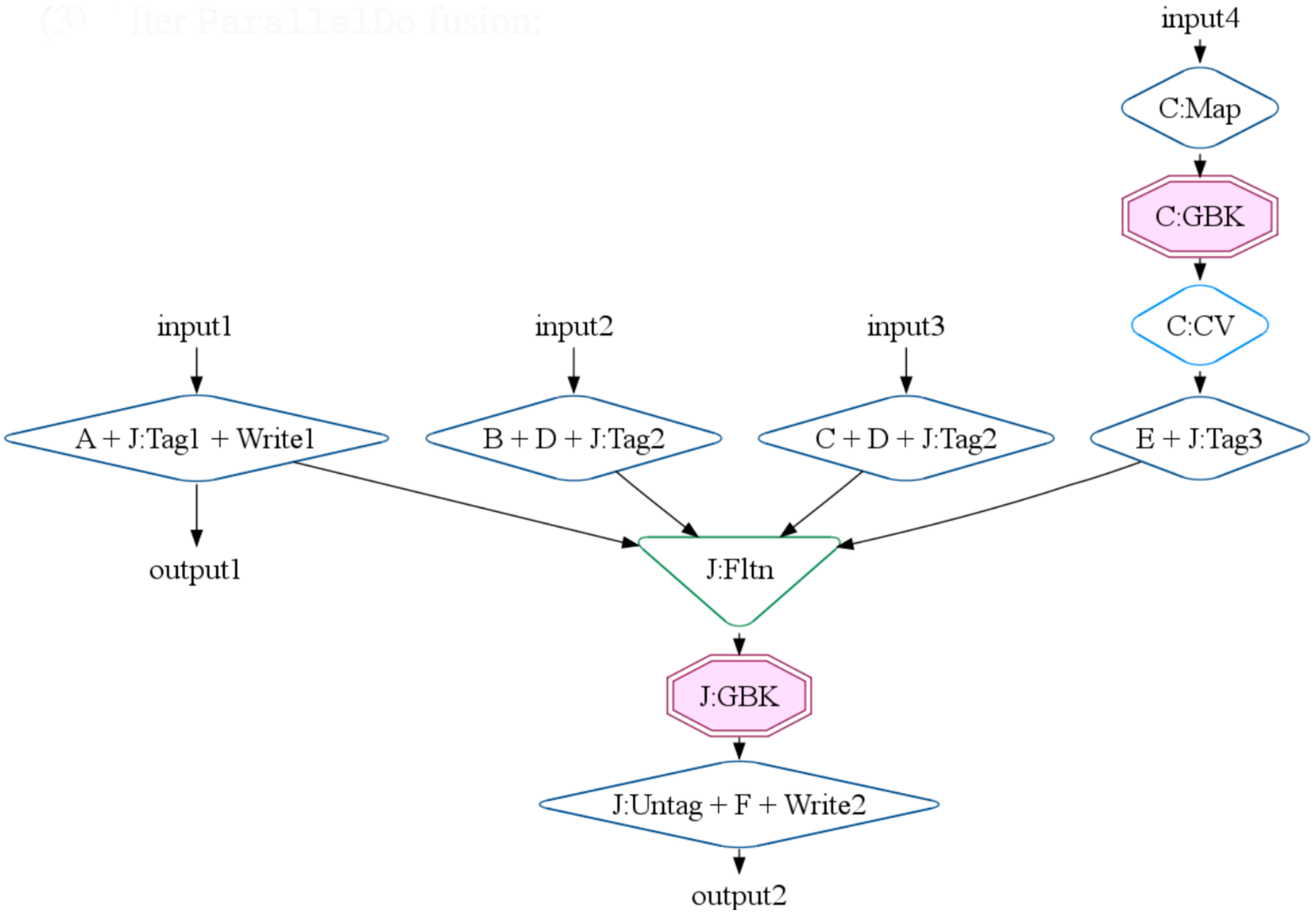
what: the two ParallelDo operations are replaced by a single multi-output ParallelDo that computes both f and g o f.

ParallelDo Fusion: sibling fusion

when: two or more ParallelDo operations read the same input

what: 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.




input4
C:Map
C:GBK
C:CV
input1
input2
input3
A + J:Tag1 + Write1
B + D + J:Tag2
C + D + J:Tag2
E + J:Tag3
output1
J:Fltn
J:GBK
J:Untag + F + Write2
output2

MapShuffleCombineReduce (MSCR)

MapShuffleCombineReduce (MSCR)

- bridges the gap between the FlumeJava API and the MapReduce primitives

MapShuffleCombineReduce (MSCR)

- bridges the gap between the FlumeJava API and the MapReduce primitives
- transforms combinations of the four primitives into single MapReduce

MapShuffleCombineReduce (MSCR)

- bridges the gap between the FlumeJava API and the MapReduce primitives
- transforms combinations of the four primitives into single MapReduce
- Generalizes MapReduce

MapShuffleCombineReduce (MSCR)

- bridges the gap between the FlumeJava API and the MapReduce primitives
- transforms combinations of the four primitives into single MapReduce
- Generalizes MapReduce
  - Multiple reducers/combiners

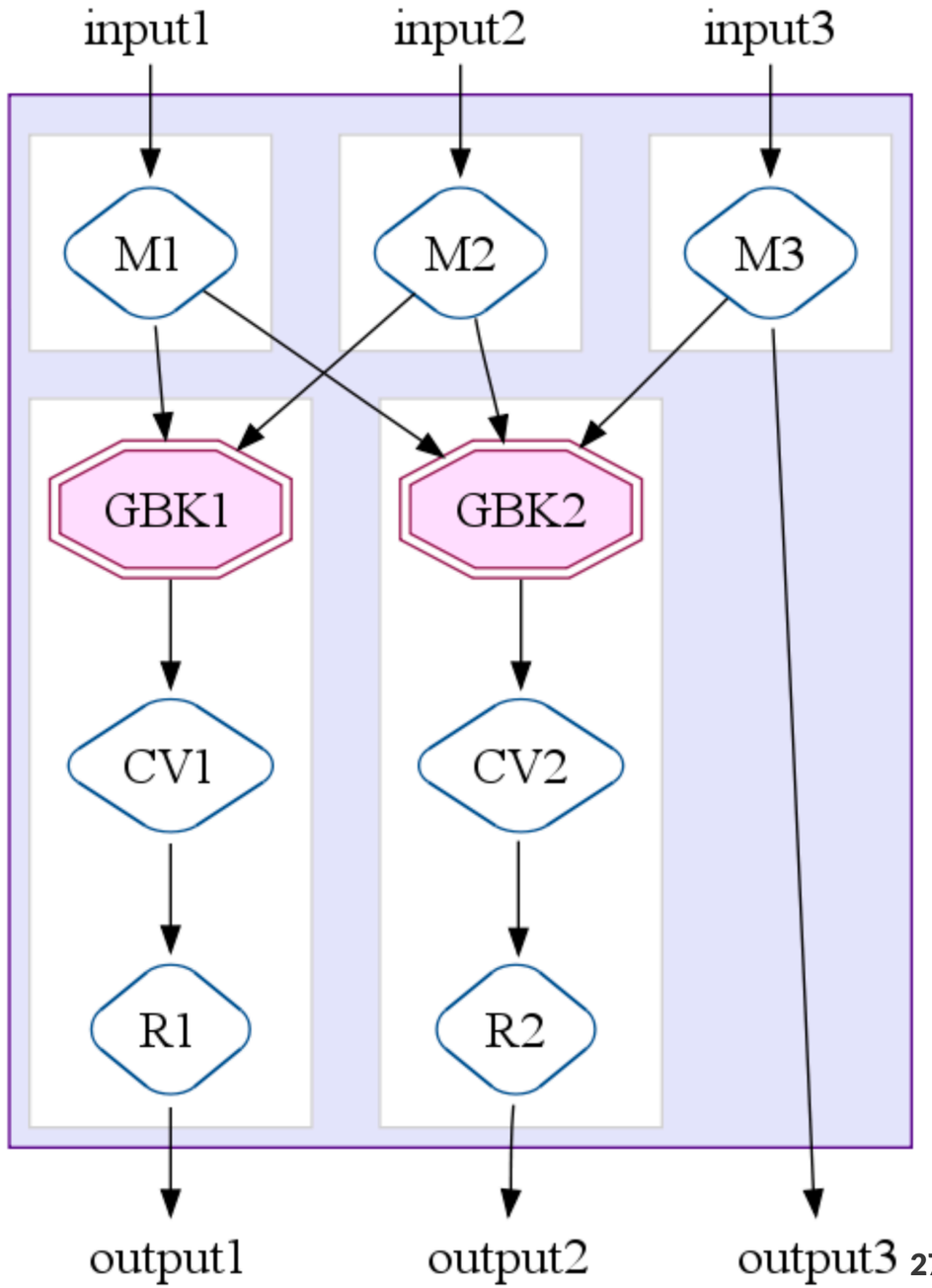
MapShuffleCombineReduce (MSCR)

- bridges the gap between the FlumeJava API and the MapReduce primitives
- transforms combinations of the four primitives into single MapReduce
- Generalizes MapReduce
  - Multiple reducers/combiners
  - Multiple output per reducer

MapShuffleCombineReduce (MSCR)

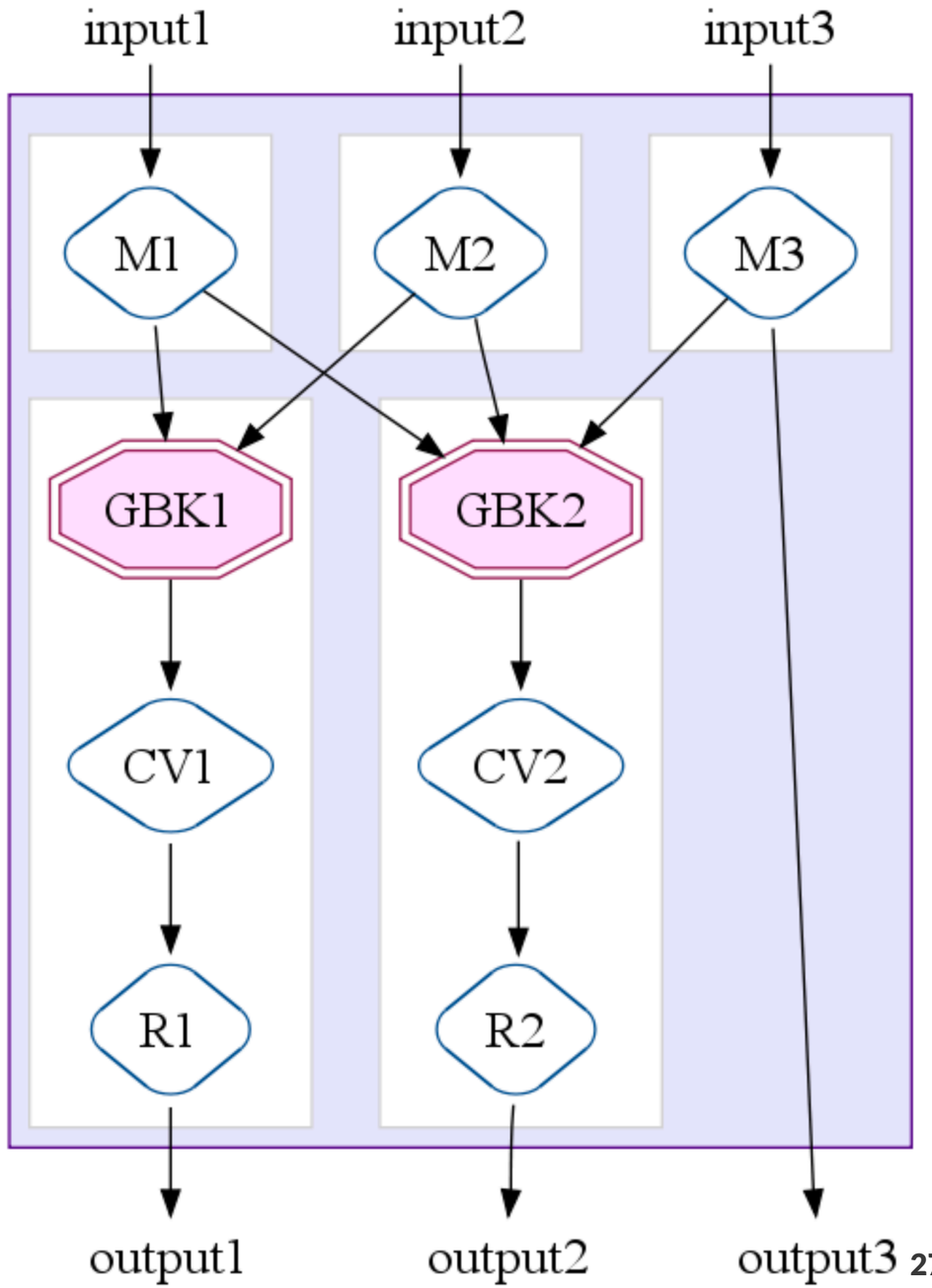
- bridges the gap between the FlumeJava API and the MapReduce primitives
- transforms combinations of the four primitives into single MapReduce
- Generalizes MapReduce
  - Multiple reducers/combiners
  - Multiple output per reducer
  - Pass-through outputs

MapShuffleCombineReduce (MSCR)



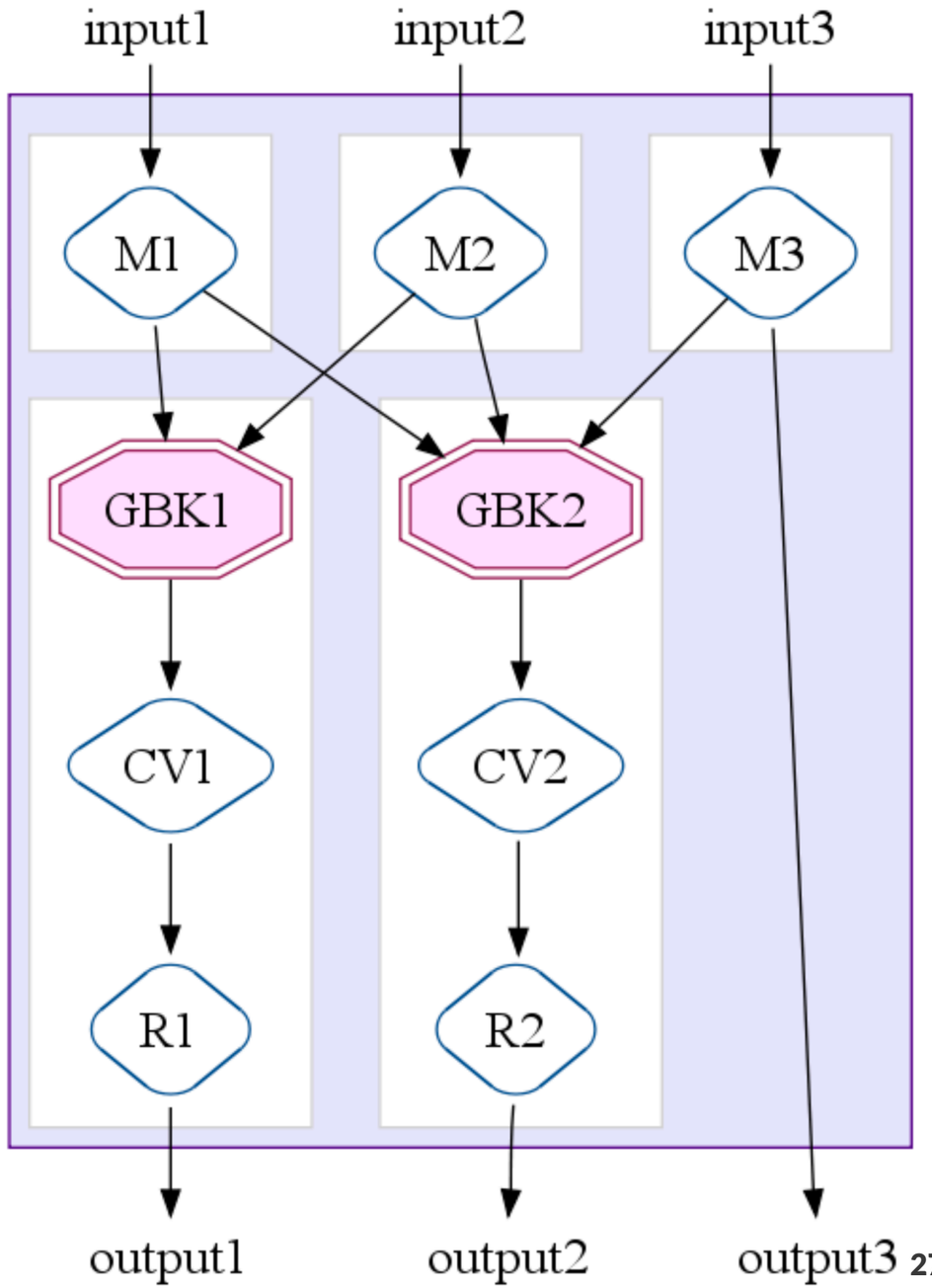
MapShuffleCombineReduce (MSCR)

- has M input channels (each performing a map operation)



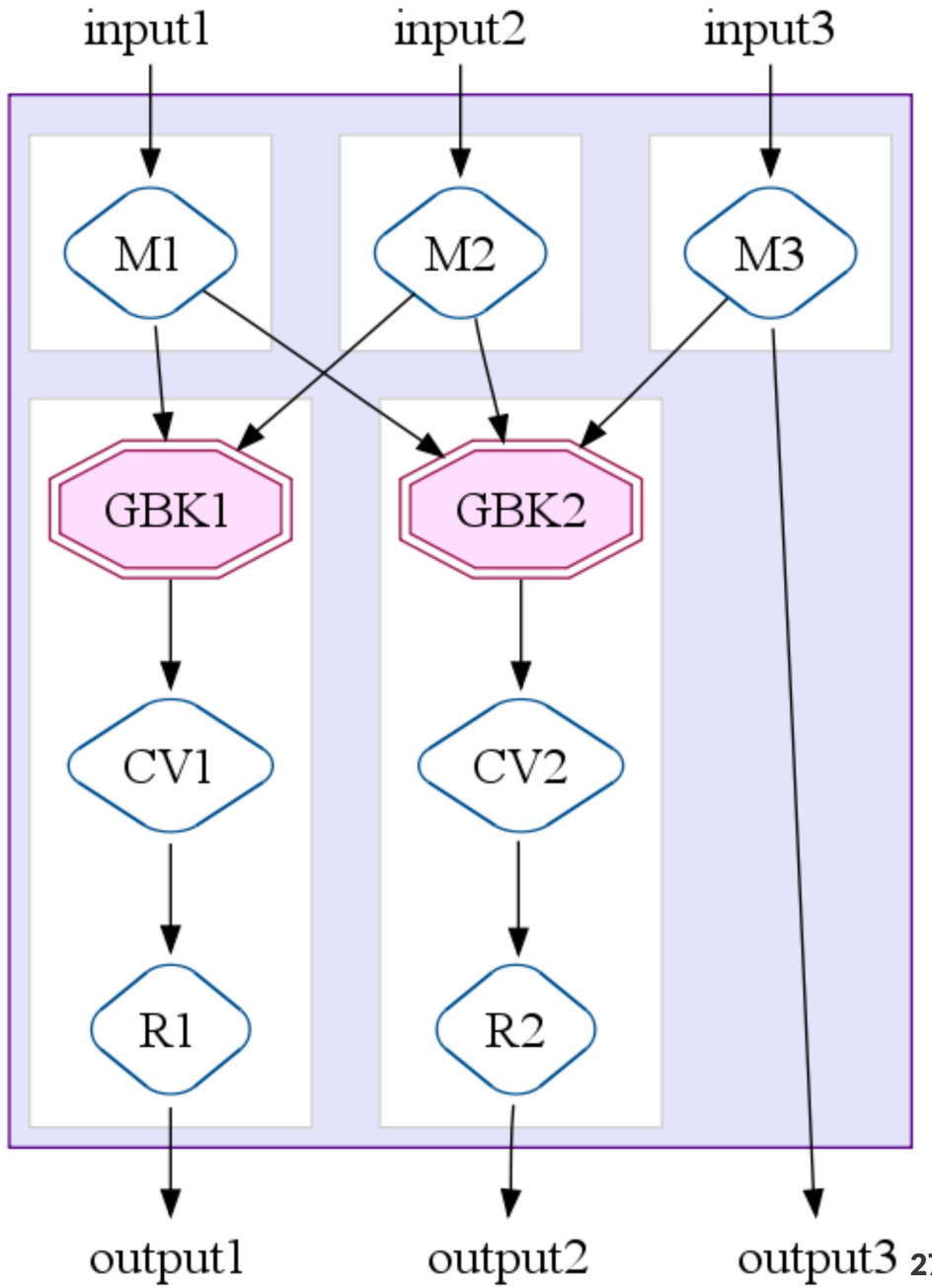
MapShuffleCombineReduce (MSCR)

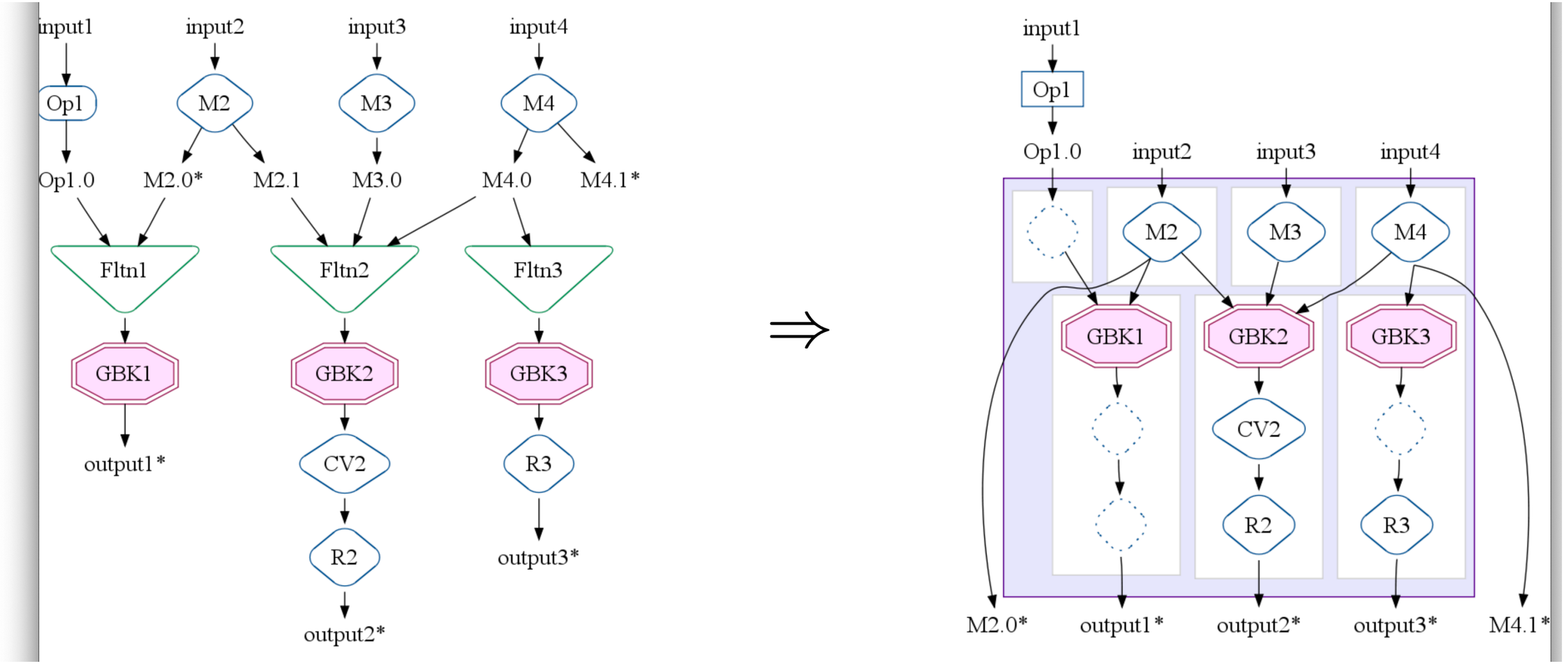
- has M input channels (each performing a map operation)
- has R output channels (each optionally performing a shuffle, an optional combine, and a reduce).



MapShuffleCombineReduce (MSCR)

- has M input channels (each performing a map operation)
- has R output channels (each optionally performing a shuffle, an optional combine, and a reduce).
- performs R “map” operation (which defaults to the identity operation) on the inputs




input1
input2
input3
input4
Op1
M2
M3
M4
Op1.0
M2.0*
M2.1
M3.0
M4.0
M4.1*
Fltn1
Fltn2
Fltn3
GBK1
GBK2
GBK3
output1*
CV2
R3
R2
output3*
output2*
input1
Op1
Op1.0
input2
input3
input4
M2
M3
M4
GBK1
GBK2
GBK3
CV2
R2
R3
M2.0*
output1*
output2*
output3*
M4.1*

Insert fusion blocks

When: If two GroupByKey operations are connected by a producer-consumer chain of one or more ParallelDo operations.

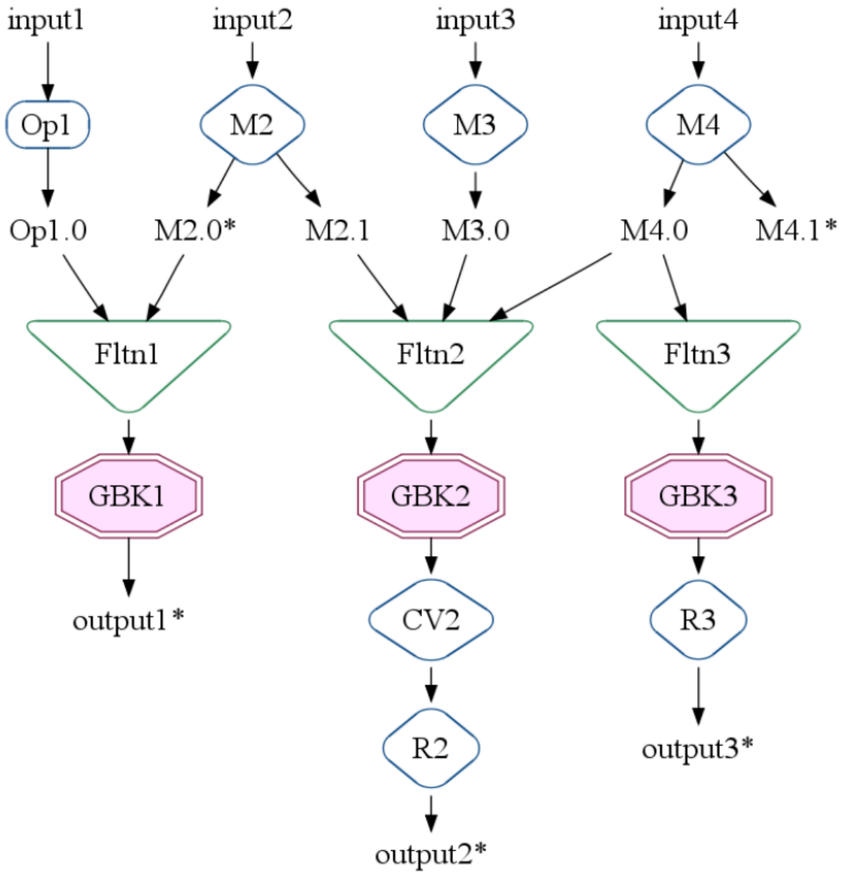
What: the optimizer must choose which ParallelDos should fuse "up" into the output channel of the earlier GroupByKey, and which should fuse "down" into the input channel of the later GroupByKey

How: The optimizer estimates the size of the intermediate PCollections along the chain of ParallelDos and marks it as boundary **blocking** ParallelDo fusion.

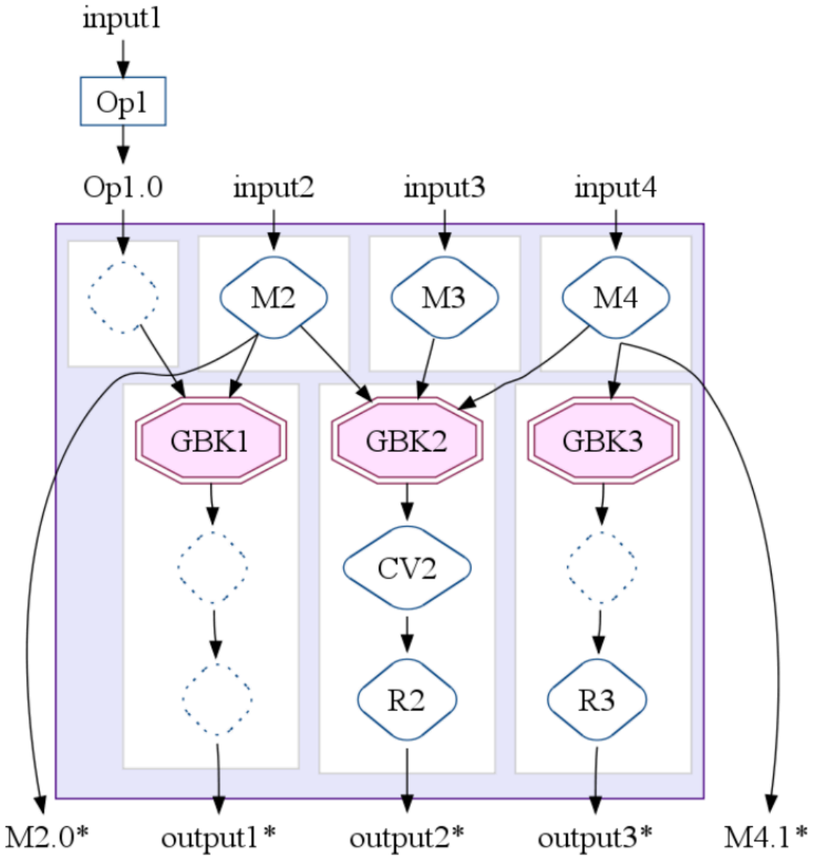
MSCR Fusion

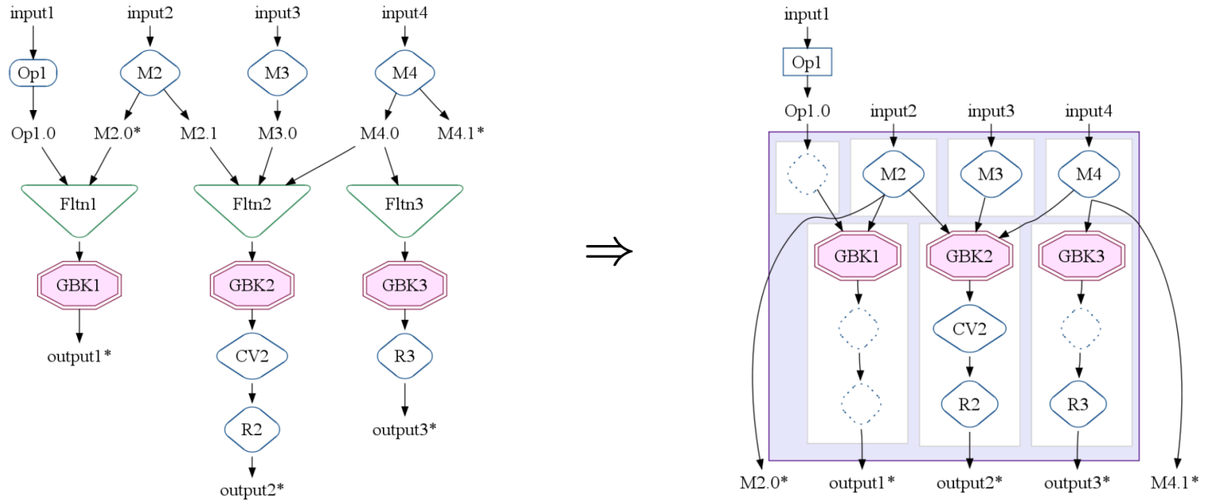
An MSCR operation is produced from a set of related GroupByKey operations.

In turn, GroupByKey operations are related if they consume the same inputs created by the same (fused) ParallelDo operations.

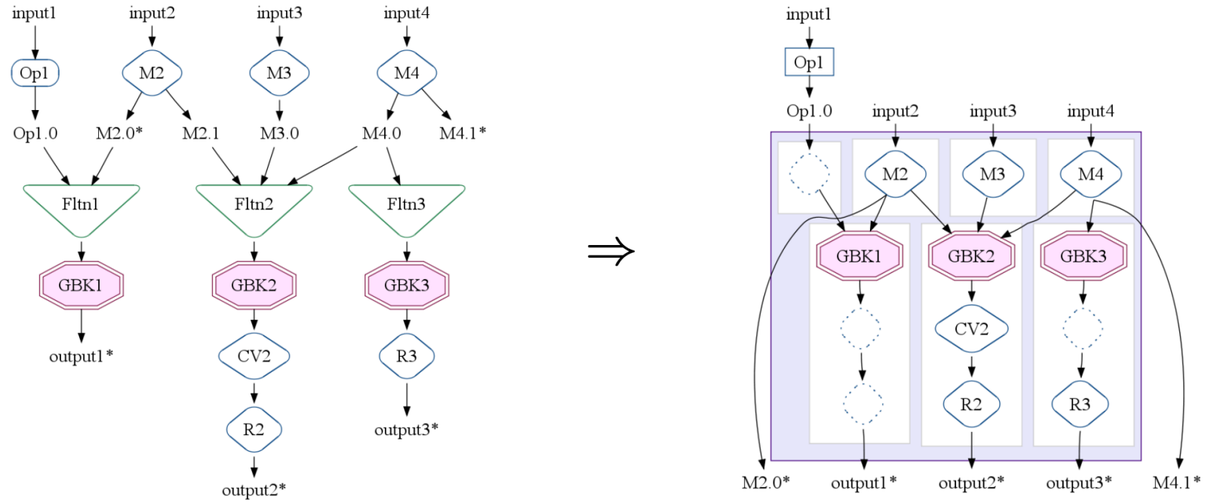

input1
input2
input3
input4
Op1
M2
M3
M4
Op1.0
M2.0*
M2.1
M3.0
M4.0
M4.1*
Fltn1
Fltn2
Fltn3
GBK1
GBK2
GBK3
output1*
CV2
R3
R2
output3*
output2*

⇒

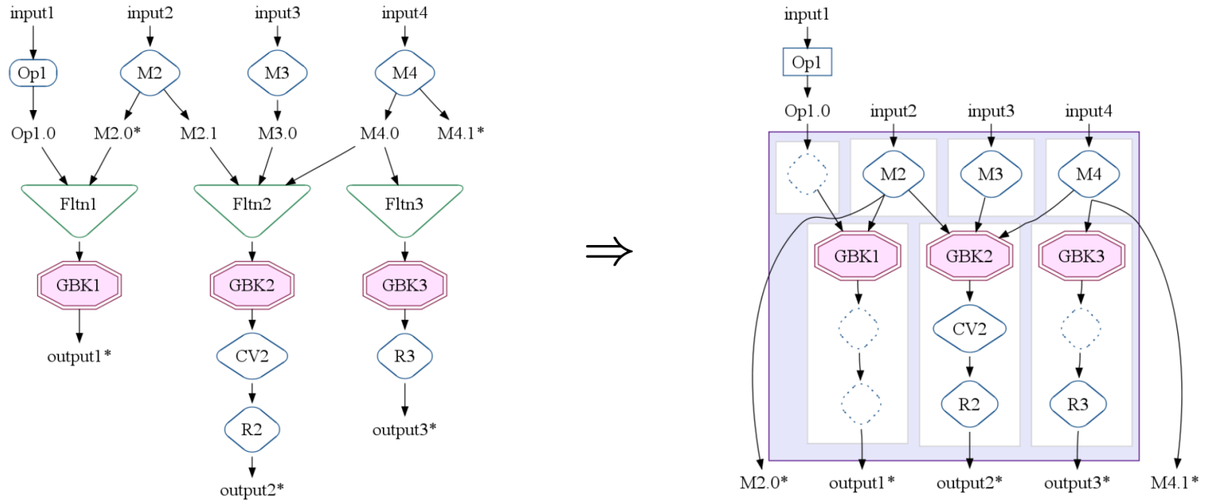

input1
Op1
Op1.0
input2
input3
input4
M2
M3
M4
GBK1
GBK2
GBK3
CV2
R2
R3
M2.0*
output1*
output2*
output3*
M4.1*


input1
input2
input3
input4
Op1
M2
M3
M4
Op1.0
M2.0*
M2.1
M3.0
M4.0
M4.1*
Fltn1
Fltn2
Fltn3
GBK1
GBK2
GBK3
output1*
CV2
R3
R2
output3*
output2*
input1
Op1
Op1.0
input2
input3
input4
M2
M3
M4
GBK1
GBK2
GBK3
CV2
R2
R3
M2.0*
output1*
output2*
output3*
M4.1*

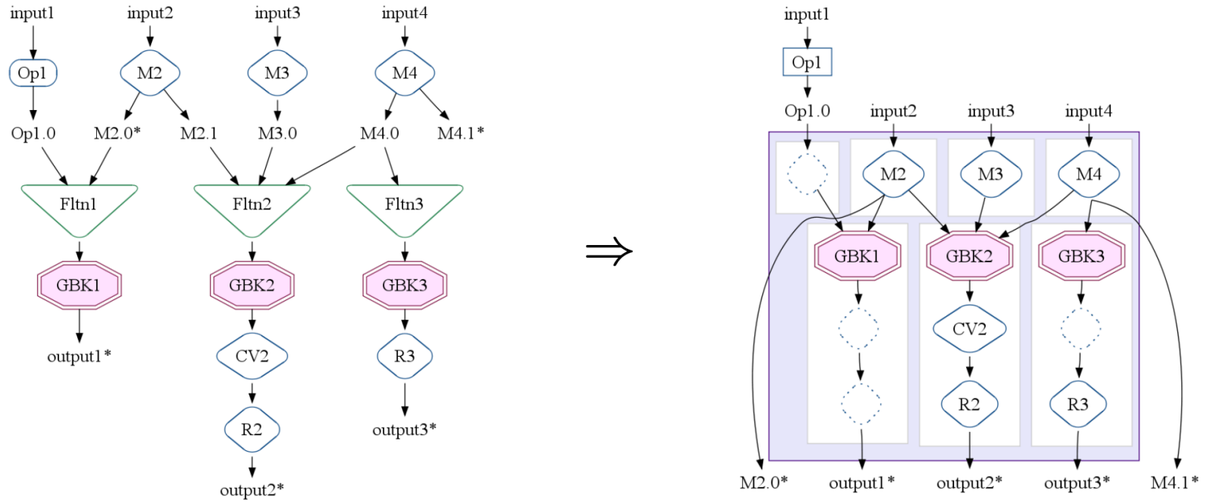
- all three GroupByKey operations are related, and hence seed a single MSCR operation.



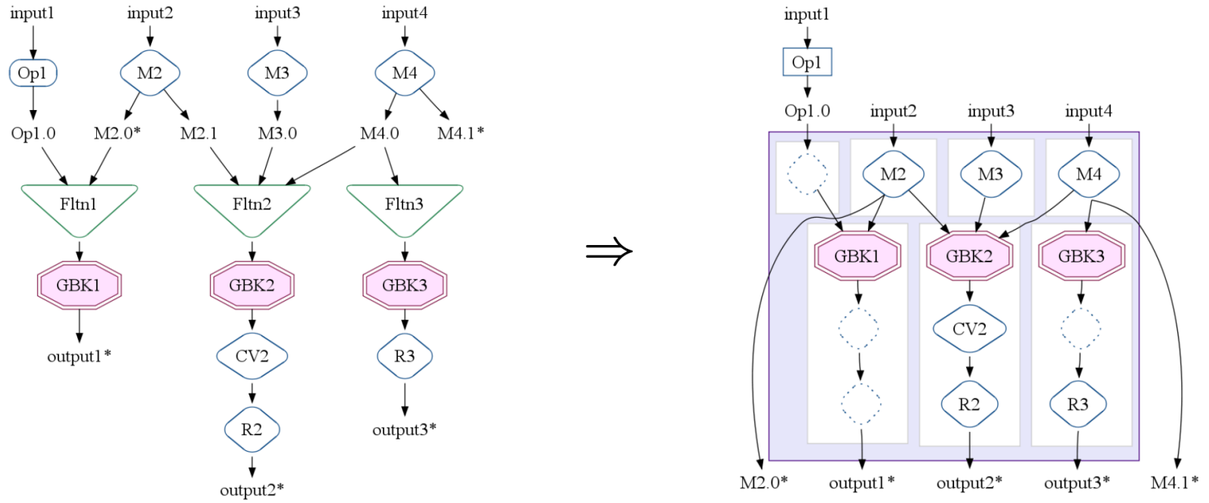
- all three GroupByKey operations are related, and hence seed a single MSCR operation.
  - GBK1 is related to GBK2 because they both consume outputs of ParallelDo M2.



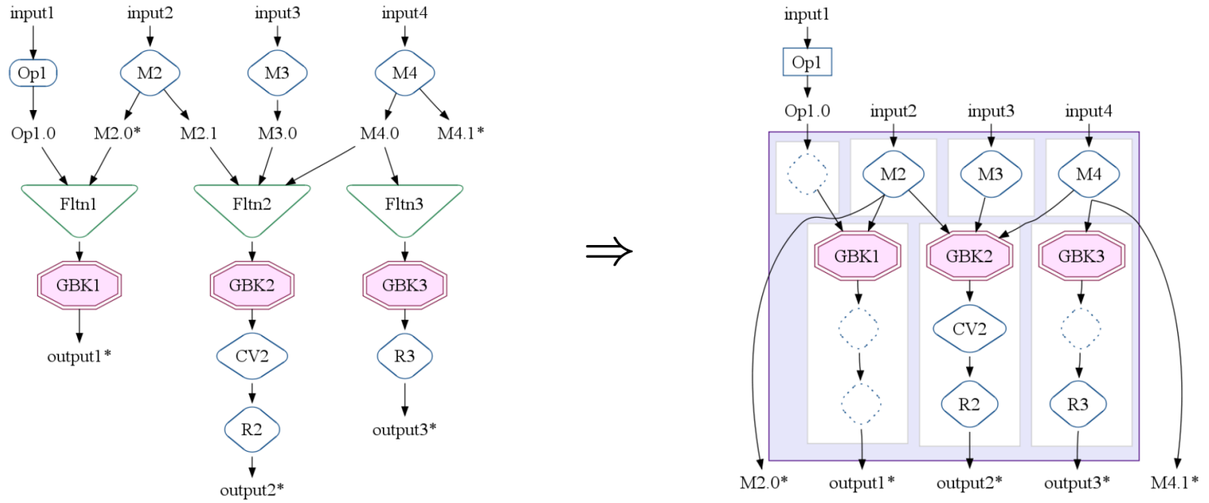
- all three GroupByKey operations are related, and hence seed a single MSCR operation.
  - GBK1 is related to GBK2 because they both consume outputs of ParallelDo M2.
  - GBK2 is related to GBK3 because they both consume PCollection M4.0.

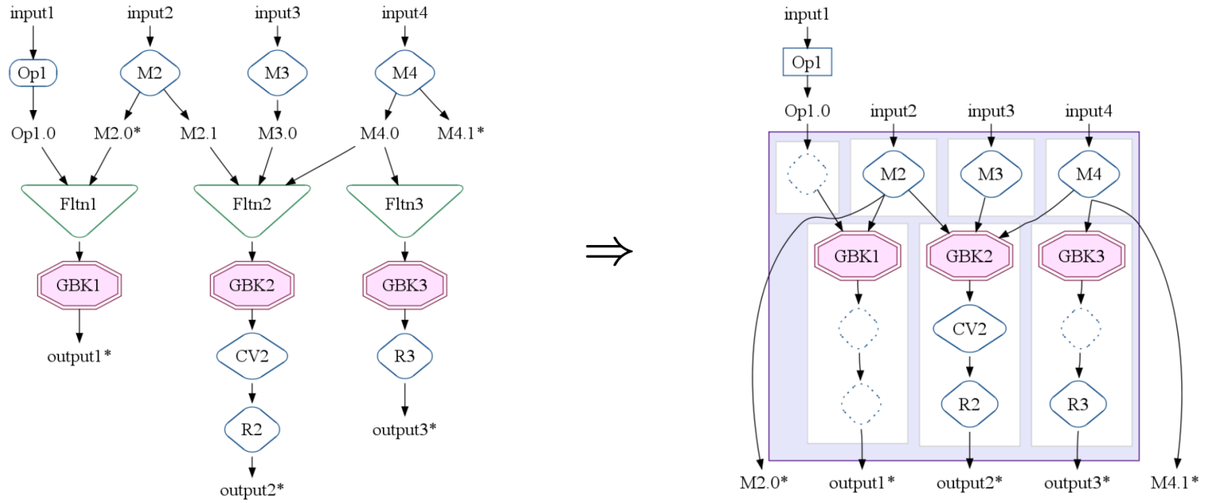


- all three GroupByKey operations are related, and hence seed a single MSCR operation.
  - GBK1 is related to GBK2 because they both consume outputs of ParallelDo M2.
  - GBK2 is related to GBK3 because they both consume PCollection M4.0.
  - The ParallelDos M2, M3, and M4 are incorporated as MSCR input channels.

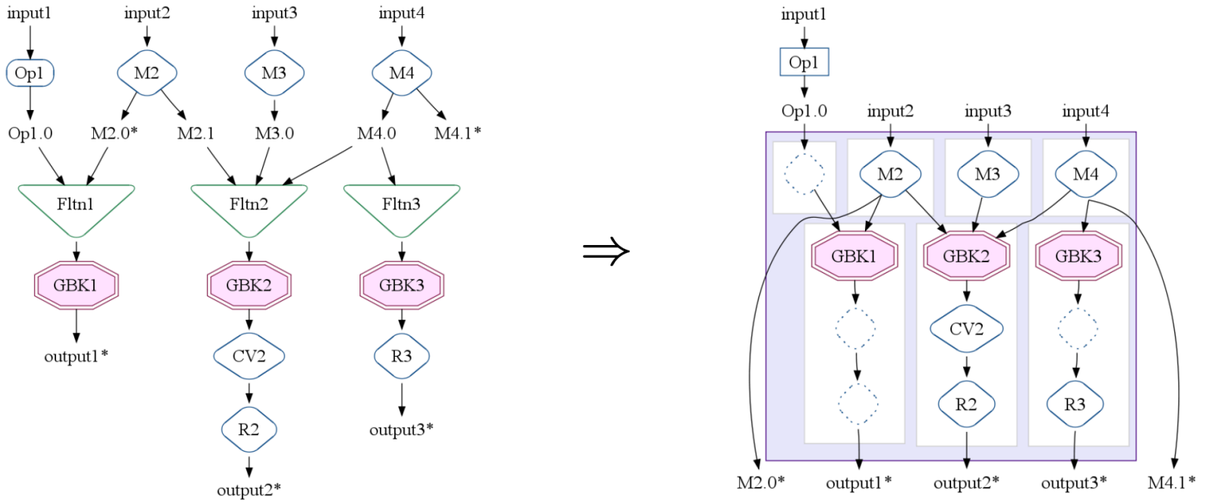


- all three GroupByKey operations are related, and hence seed a single MSCR operation.
  - GBK1 is related to GBK2 because they both consume outputs of ParallelDo M2.
  - GBK2 is related to GBK3 because they both consume PCollection M4.0.
  - The ParallelDos M2, M3, and M4 are incorporated as MSCR input channels.
- Each of the GroupByKey operations becomes a grouping output channel.

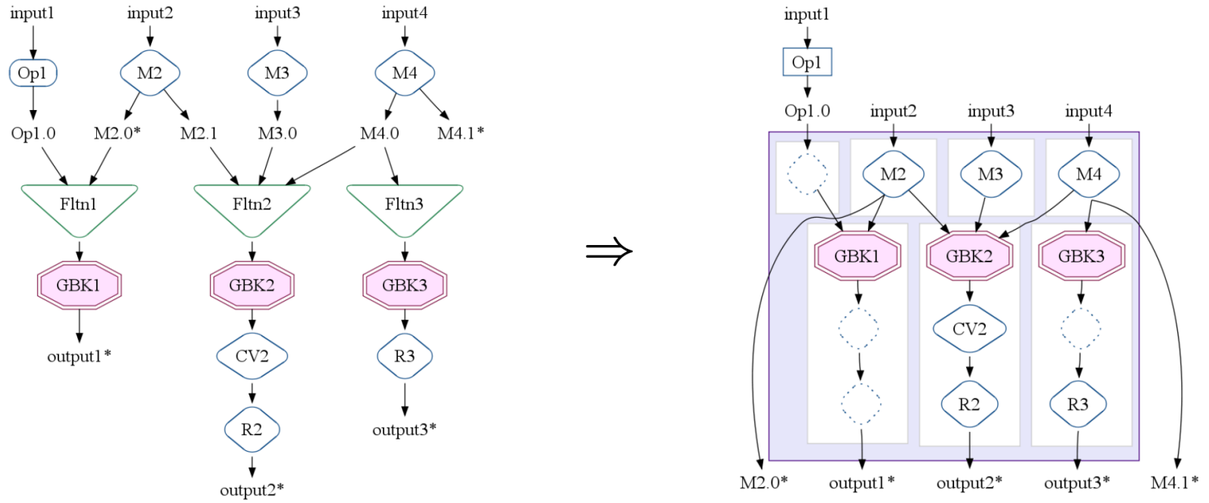



input1
input2
input3
input4
Op1
M2
M3
M4
Op1.0
M2.0*
M2.1
M3.0
M4.0
M4.1*
Fltn1
Fltn2
Fltn3
GBK1
GBK2
GBK3
output1*
CV2
R3
R2
output3*
output2*
input1
Op1
Op1.0
input2
input3
input4
M2
M3
M4
GBK1
GBK2
GBK3
CV2
R2
R3
M2.0*
output1*
output2*
output3*
M4.1*

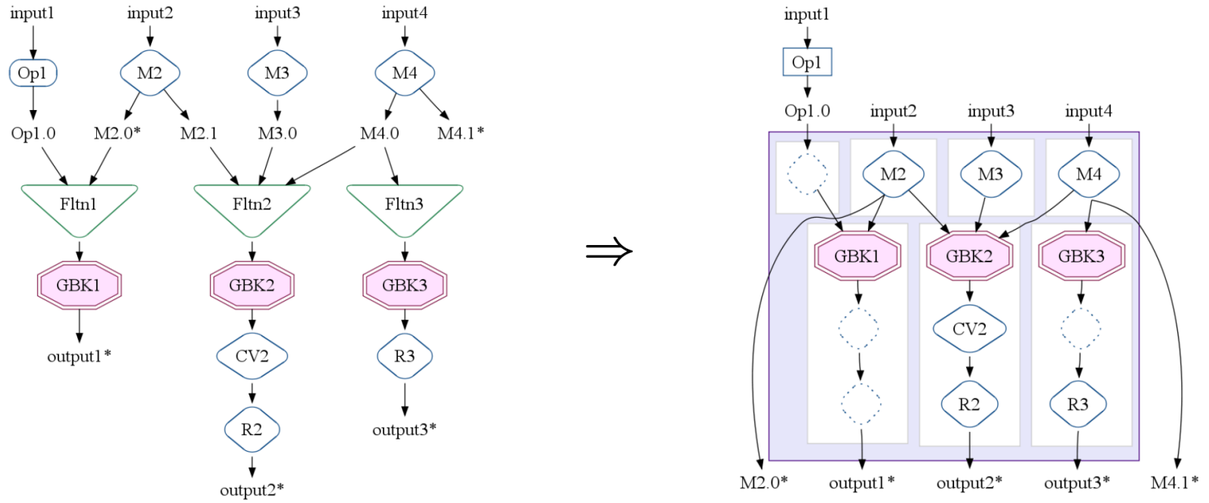
- GBK2's output channel incorporates the CV2 CombineValues operation.



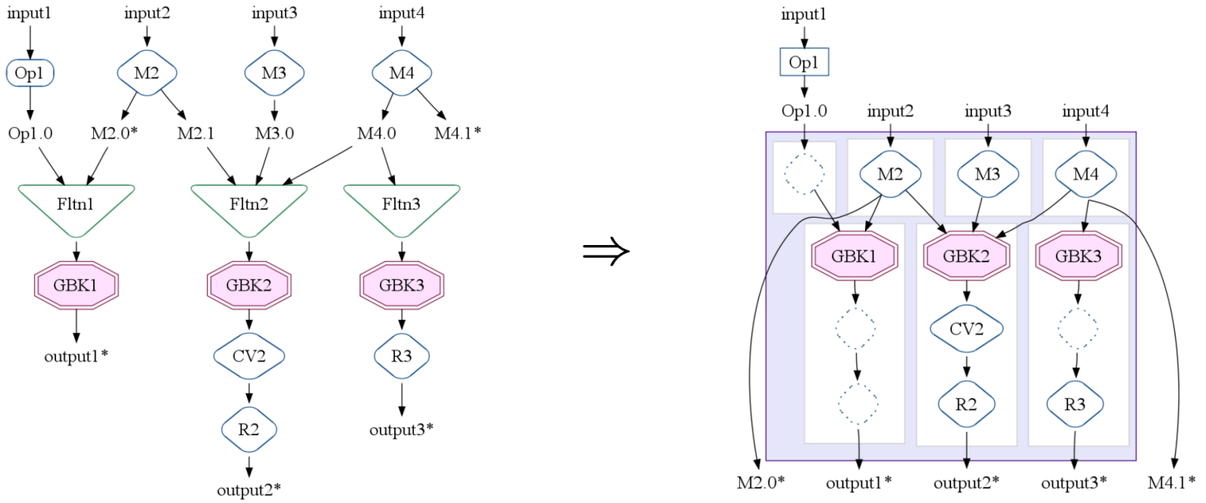
- GBK2's output channel incorporates the CV2 CombineValues operation.
- The R2 and R3 ParallelDos are also incorporated into output channels.



- 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.



- 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.



Workflow Engines

Workflow Engines

- Task-driven, they decouple the task management from the task process.

Workflow Engines

- Task-driven, they decouple the task management from the task process.
- They have limited knowledge of the data the task is processing.

Workflow Engines

- Task-driven, they decouple the task management from the task process.
- They have limited knowledge of the data the task is processing.
- Python-based, they aim at democratizing the ability to orchestrate jobs.

A note on Data Orchestration

source

Apache Airflow

Apache
Airflow

What Airflow is...

Apache Airflow is a dataflow orchestrator to define data engineering workflows

Airflow allows to parallelize jobs, schedule them appropriately with dependencies and historically reprocess data when needed.

Airflow glues all the data engineering steps, i.e., ingest, transform, store.

What Airflow is not...

A data processing tool. No processing happens in airflow, so there's no ~~need for~~ fault tolerance

Airflow does not provide any built-in workflow versioning

Airflow does not support streaming computation (talk later)[1]

[1] The reason as to why Airflow does not support streaming is that there are no obvious behavior rules that could be set so that the airflow scheduler could deterministically check if it has been completed or not.

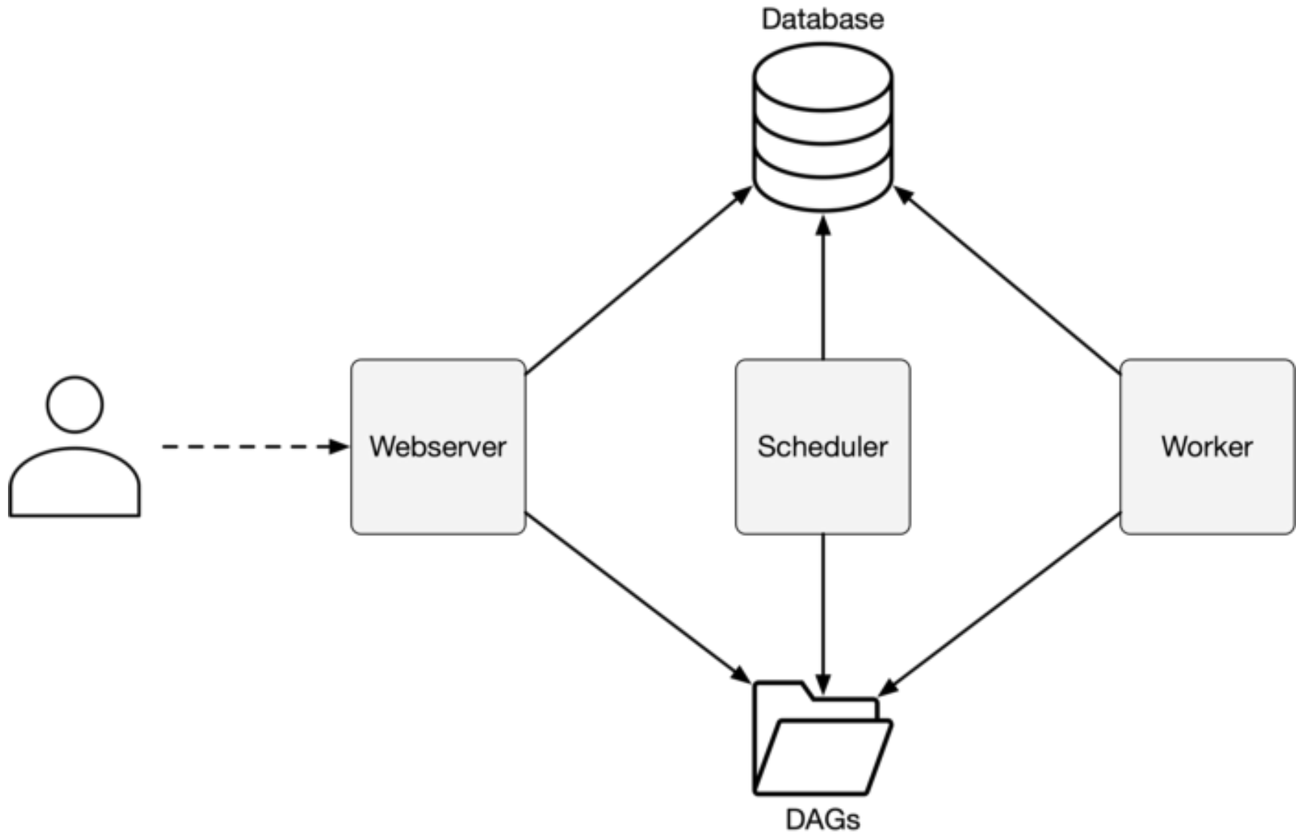
The DAG

Airflow (and almost any orchestrato), organizes the processing in a Directed Acyclic Graph (DAG). Acyclicity is required to reduce ambiguity.

Data engineering tasks are the graph nodes, Edges represent dependency relationships, e.g., if task 1 depends on task 2, then task 2 has to succeed.

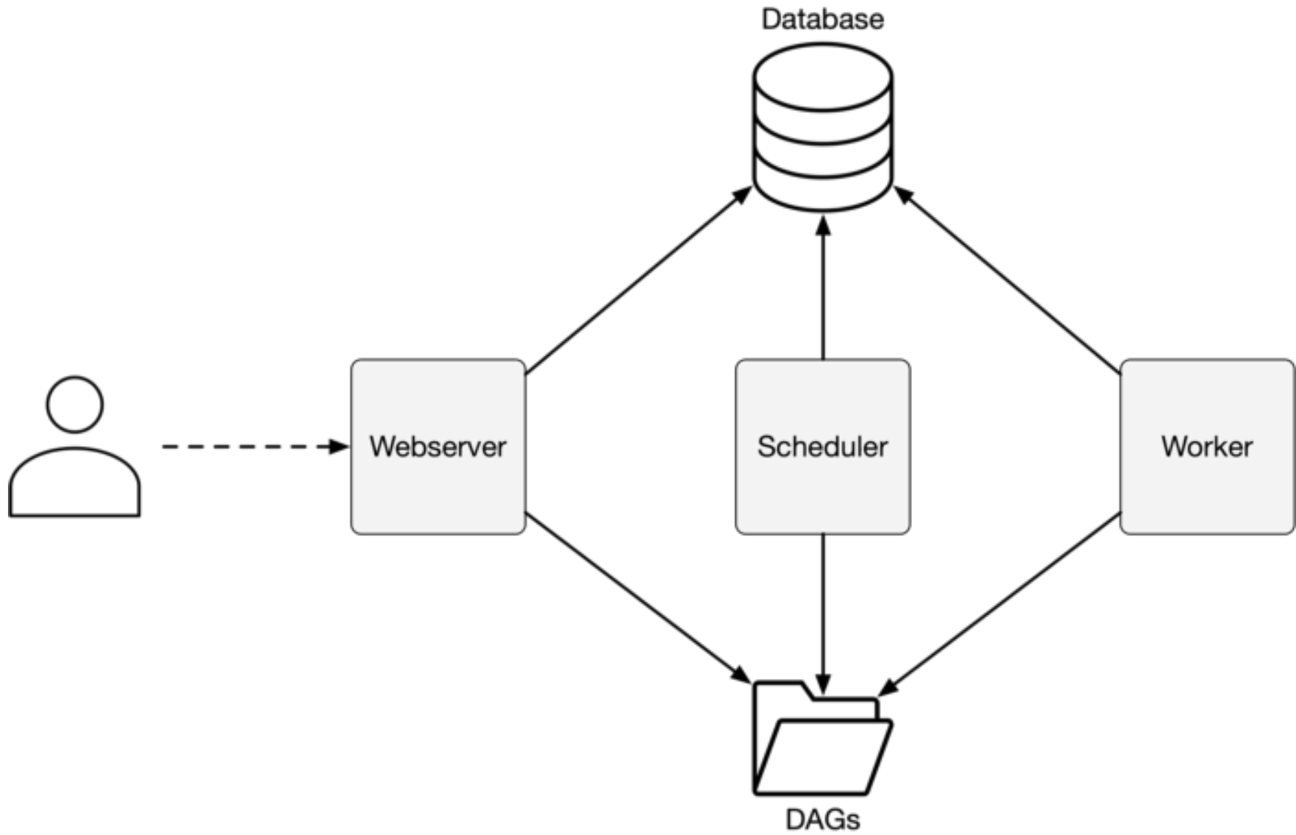
Notably, "success" doesn't have to mean that it has been succesfully completed. It can mean that it hasn't been skipped, for instance.

Airflow architecture



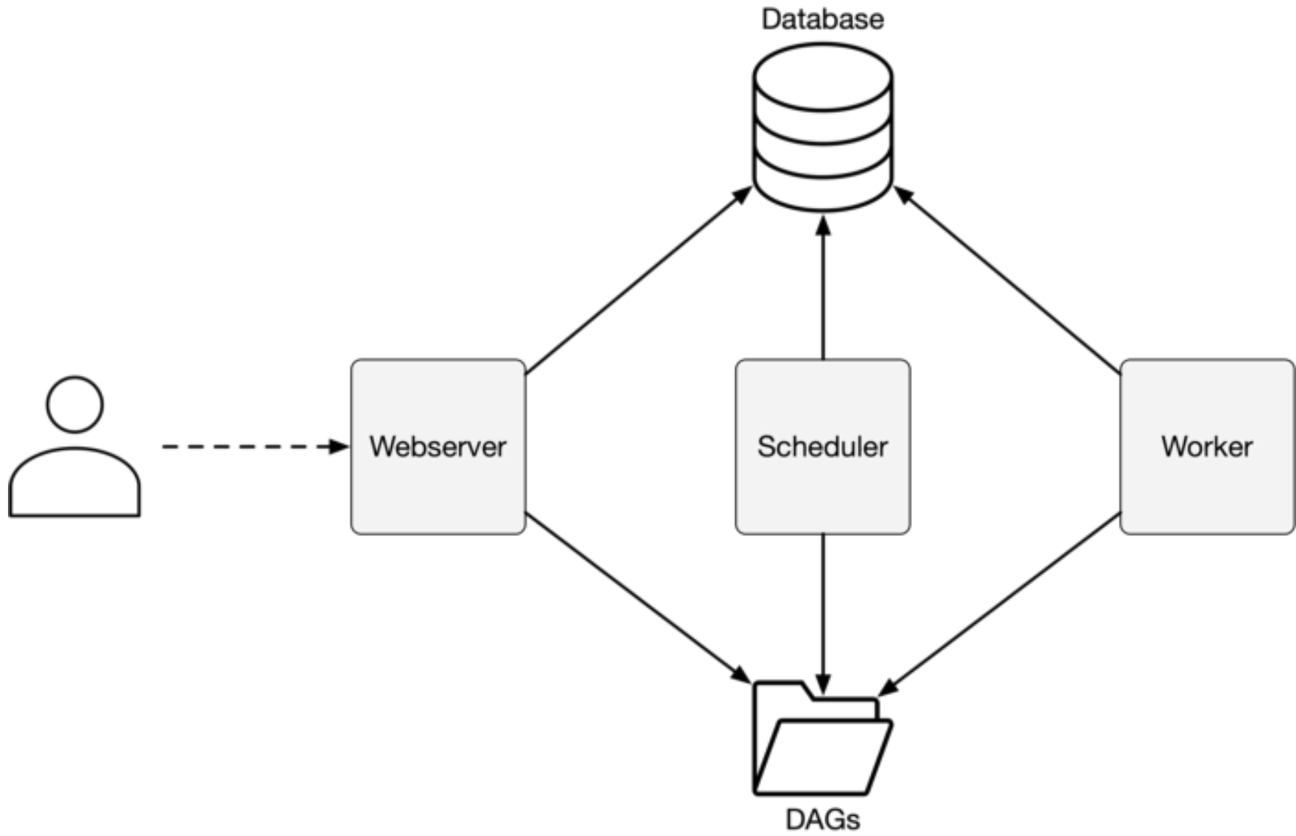
Airflow architecture

- **Web server** - nice GUI with access to the logs and most of Airflow's functionality, such as stopping and clearning DAGs.



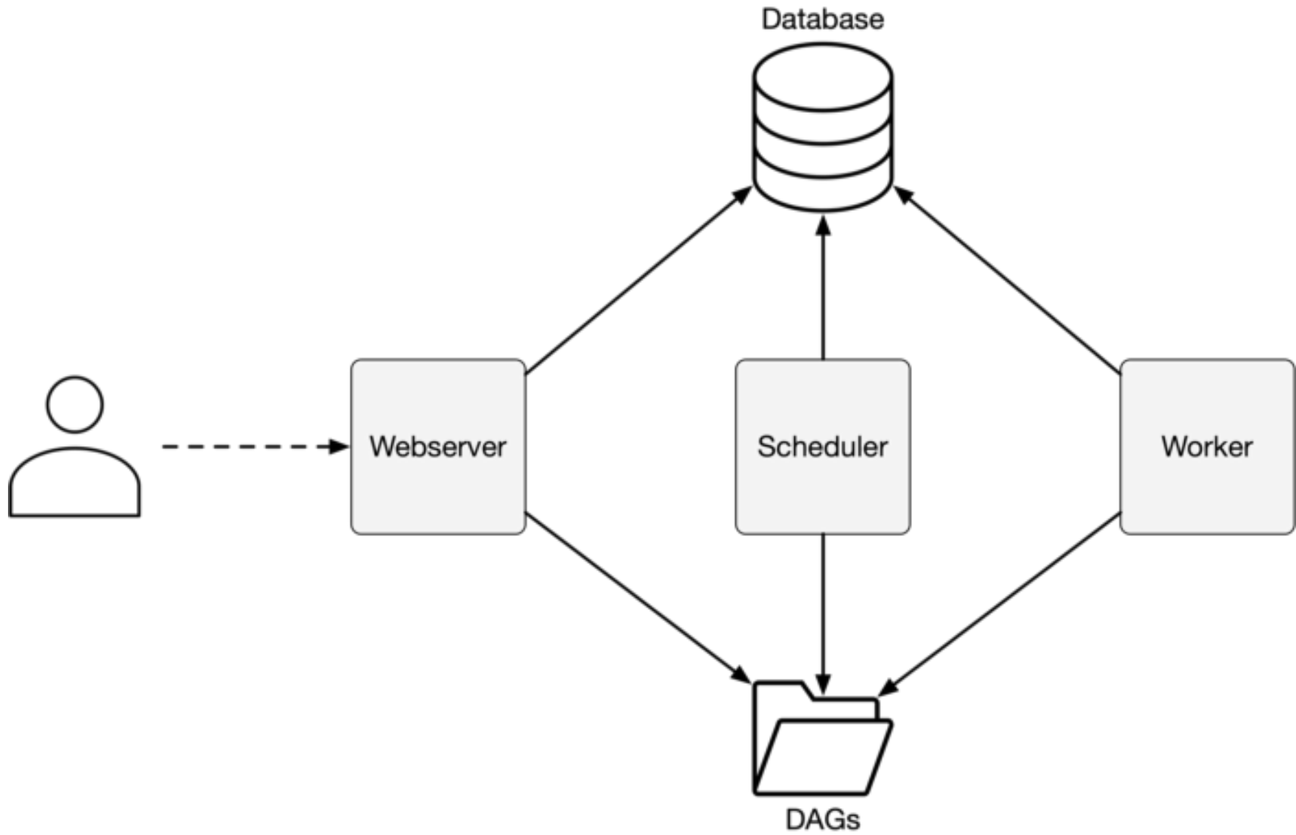
Airflow architecture

- **Web server** - nice GUI with access to the logs and most of Airflow's functionality, such as stopping and clearning DAGs.
- **Scheduler** - Puts the DAGs to action by coordinating the work to be done



Airflow architecture

- **Web server** - nice GUI with access to the logs and most of Airflow's functionality, such as stopping and clearning DAGs.
- **Scheduler** - Puts the DAGs to action by coordinating the work to be done
- **Workers** - do the job assigned by the scheduler.



Workflows

Workflows are written entirely in Python

The first thing to do is to instance the DAG, declaring a dictionary with default arguments such as *start date*, *concurrency* and *schedule interval*.

```
default_args_dict = {
    'start_date': airflow.utils.dates.days_ago(0),
    'concurrency': 1,
    'schedule_interval': None,
    'retries': 1,
    'retry_delay': datetime.timedelta(minutes=5),
}

first_dag = DAG(
    dag_id='first_dag',
    default_args=default_args_dict,
    catchup=False,
)
```

Idiosyncrasies

If you thought it would **start** at the moment of **start***date, you thought wrong~~, haha~~. The first task will actually start on start*date + schedule_interval.

In case you tried to run a DAG, for the first time, with a start date of let's say, 1 month ago, and a schedule interval of 5 minutes, then airflow would generate 8640 runs, which is the amount of 5 minute intervals within a month.

In case you didn't put `catchup=False` you will be in for a treat next time you start it, since airflow will re-run the DAG for n schedule intervals between when you turned it off and the moment of turning it on again.

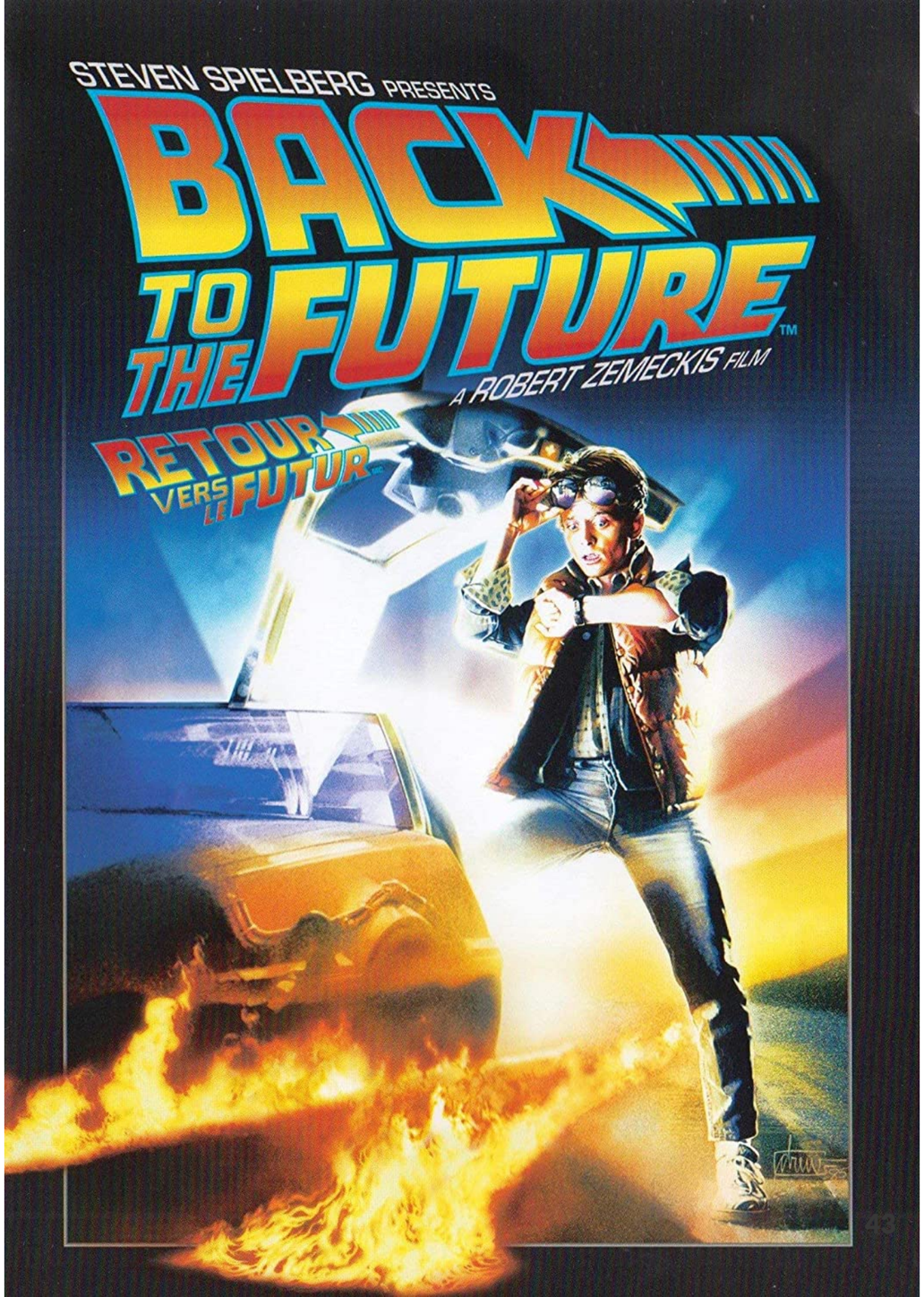
Backfilling

Often, we might desire to revisit the historical trends and movements. In such cases, we would need to compute metric and dimensions in the past,

Airflow has a recipe for Backfilling. You can call the command

```
backfill -s START_DATE -e
END_DATE dag_id
```

from the cli, and it will rerun any DAG with the specified dates, and that way you can very effortlessly satisfy your tech lead's needs.



Operators

DAGs describe **how** to run a data pipeline, operators describe **what** to do in a data pipeline.

Operators trigger data transformations, which corresponds to the **T**ransform step

Operators: Sensors, Operators, and Transfers

Operators: Sensors, Operators, and Transfers

- **Sensors**: waits for a certain time, external file, or upstream data source

Operators: Sensors, Operators, and Transfers

- **Sensors**: waits for a certain time, external file, or upstream data source
- **Operators**: triggers a certain action (e.g. run a bash command, execute a python function, or execute a Hive query, etc)

Operators: Sensors, Operators, and Transfers

- **Sensors**: waits for a certain time, external file, or upstream data source
- **Operators**: triggers a certain action (e.g. run a bash command, execute a python function, or execute a Hive query, etc)
- **Transfers**: moves data from one location to another

DummyOperator (1/2)

The simplest, yet not useless, operator that there is, is the dummy:

```
task_one = DummyOperator(
    task_id='get_spreadsheet',
    dag=first_dag,
    depends_on_past=False,
)
```

Operators are declared similarly as dags, and only require two arguments at first:

DummyOperator (1/2)

The simplest, yet not useless, operator that there is, is the dummy:

```
task_one = DummyOperator(
    task_id='get_spreadsheet',
    dag=first_dag,
    depends_on_past=False,
)
```

Operators are declared similarly as dags, and only require two arguments at first:

- `task_id` = unique id for the operator

DummyOperator (1/2)

The simplest, yet not useless, operator that there is, is the dummy:

```
task_one = DummyOperator(
    task_id='get_spreadsheet',
    dag=first_dag,
    depends_on_past=False,
)
```

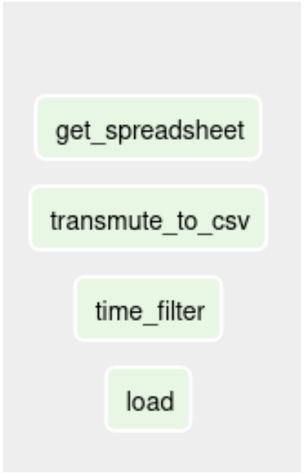
Operators are declared similarly as dags, and only require two arguments at first:

- `task_id` = unique id for the operator
- `dag` = which dag does does this operator belong to? Because there can be sub-dags, this needs to be declared explicitly.

DummyOperator (2/2)

In practice, the dummy operator does nothing, but it can be used to do two things:

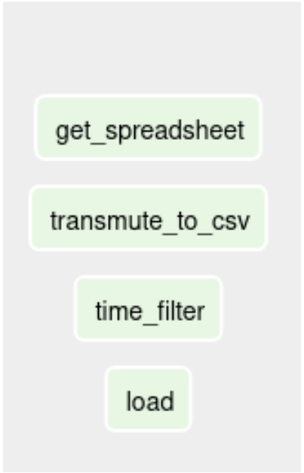
More often than not we would like to visualize an ETL process in the least time-wasting manner as possible.



DummyOperator (2/2)

In practice, the dummy operator does nothing, but it can be used to do two things:

- sketching a dag out



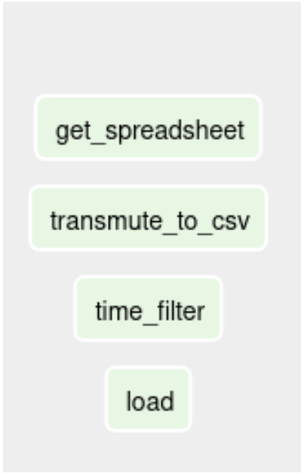
More often than not we would like to visualize an ETL process in the least time-wasting manner as possible.

DummyOperator (2/2)

In practice, the dummy operator does nothing, but it can be used to do two things:

- sketching a dag out
- trigger rule sorcery

More often than not we would like to visualize an ETL process in the least time-wasting manner as possible.



Non Dummy Operators

Most of airflow's operators function in a similar way.

Non Dummy Operators

Most of airflow's operators function in a similar way.

- `PostgreSQL` allows you to communicate with postgres instances.

Non Dummy Operators

Most of airflow's operators function in a similar way.

- `PostgreSQL` allows you to communicate with postgres instances.
- `BashOperator` allows you to run shell scripts

Non Dummy Operators

Most of airflow's operators function in a similar way.

- `PostgreSQL` allows you to communicate with postgres instances.
- `BashOperator` allows you to run shell scripts
- `PythonOperator` allows you to run Python code

BashOperator

Let us define a non-dummy operator, that runs a bash command:

```
task_one = BashOperator(
    task_id='get_spreadsheet',
    dag=first_dag,
    bash_command="curl http://www.gerbode.net/spreadsheet.xlsx --output /usr/local/airflow/data/{{ds_nodash}}.xlsx",
    trigger_rule='all_success',
    depends_on_past=False

)
```

BashOperator

Let us define a non-dummy operator, that runs a bash command:

```
task_one = BashOperator(
    task_id='get_spreadsheet',
    dag=first_dag,
    bash_command="curl http://www.gerbode.net/spreadsheet.xlsx --output /usr/local/airflow/data/{{ds_nodash}}.xlsx",
    trigger_rule='all_success',
    depends_on_past=False

)
```

- `bash_command`- in case you know bash, you have already figured it out that this works almost exactly as typing `bash -c "some command"`

BashOperator

Let us define a non-dummy operator, that runs a bash command:

```
task_one = BashOperator(
    task_id='get_spreadsheet',
    dag=first_dag,
    bash_command="curl http://www.gerbode.net/spreadsheet.xlsx --output /usr/local/airflow/data/{{ds_nodash}}.xlsx",
    trigger_rule='all_success',
    depends_on_past=False

)
```

- `bash_command`- in case you know bash, you have already figured it out that this works almost exactly as typing `bash -c "some command"`
- `depends_on_past=False` means that if it failed during the previous dag run then it won't run this time. This is important because, as we've seen, airflow can have overlapping dag runs.

Python Operator

The last part we have to go over is both the briefest and most important, the Python Operator:

Very similar to the bash operator, having `task_id`, `dag`, `trigger_rule` and `depends_on_past`, as pretty much any other operator.

Python Operator takes a function as an input `python_callable`, that is where the python part of the python operator lies at. The callable's arguments are given by the `op_kwargs` dictionary, which also allows you to template these arguments.

```
task_one = PythonOperator(
    task_id='get_spreadsheet',
    dag=second_dag,
    python_callable=_get_spreadsheet,
    op_kwargs={
        "output_folder": "/usr/local/airflow/data",
        "epoch": "{{ execution_date.int_timestamp }}",
        "url": "http://www.gerbode.net/spreadsheet.xlsx"
    },
    trigger_rule='all_success',
    depends_on_past=False,
)
```

Task Relationships

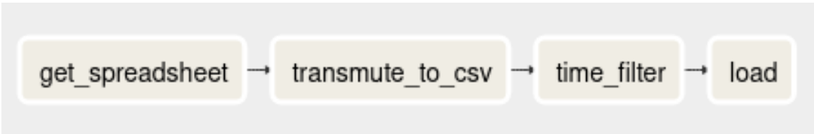
The directed relationships are given by the edges, and are very easily defined as follows, written at the end of the DAG file:

```
node_1 >> node_2 >> node_3 >> node_4
```

Looks like

Example

We want to show that the ETL process starts with `get_spreadsheet`, then its format is transmuted to a proper csv on `transmute_to_csv`, which is then filtered by time, `time_filter` to be loaded somewhere, `load`.



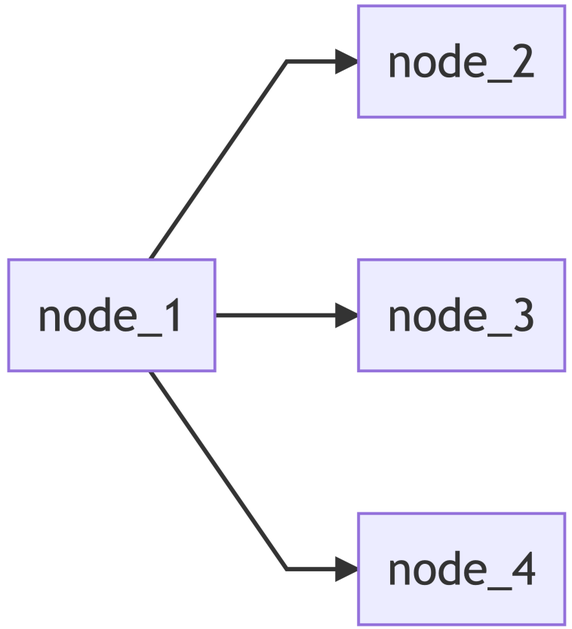

```
get_spreadsheet >> transmute_to_csv >> time_filter >> load
```

Will yield the following sequential configuration:

N-Ary Relationships

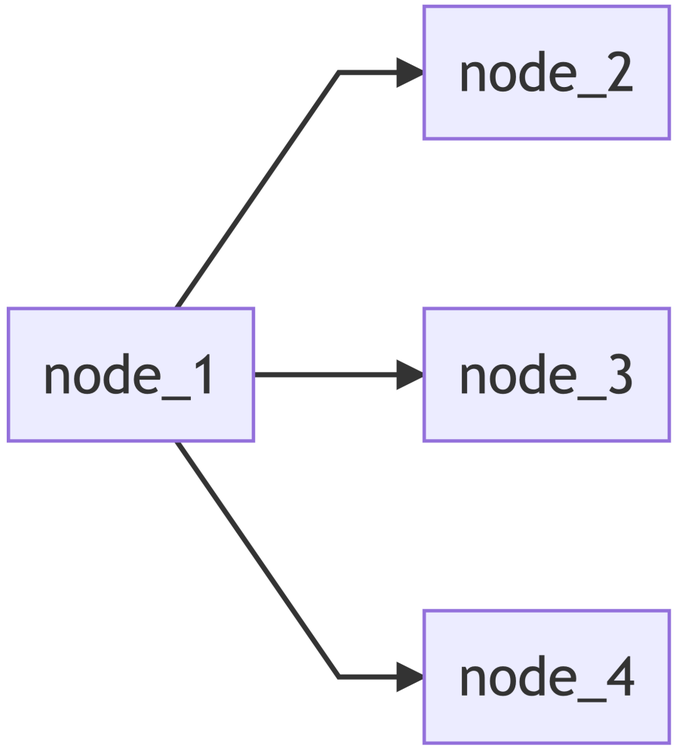
many-to-one or one-to-many relationships are encoded as lists, so you can put to code that many nodes have an edge to one node as follows:

```
[node_1, node_2, node_3] >> node_4
```



and one to many:

```
node_1 >> [node_2, node_3, node_4]
```



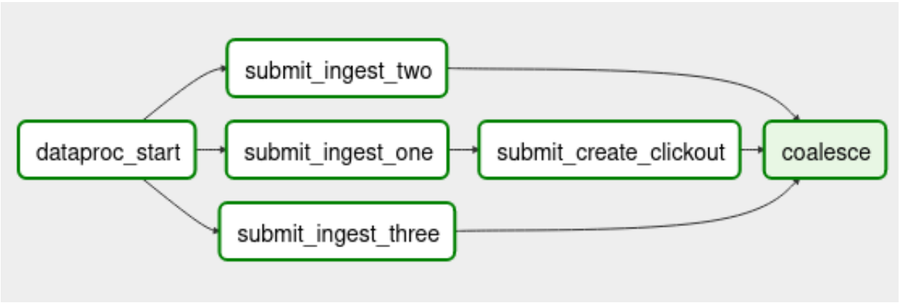
Example

Let's say that you have to create an ETL process entirely done in the cloud,
and we define the following tasks:

dataproc*start* >> [*dataproc*ingest*one*, *dataproc*ingest*two*, *dataproc*ingest_three]

dataproc*ingest*one >> dataproc*create*clickout

[dataproc*create*clickout, dataproc*ingest*two, dataproc*ingest*three] >> end



Example

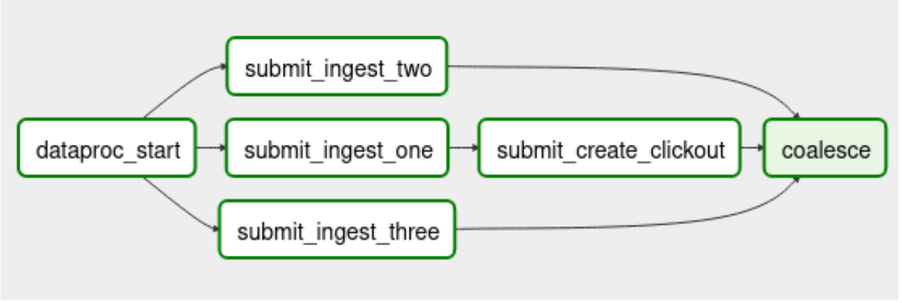
Let's say that you have to create an ETL process entirely done in the cloud, and we define the following tasks:

- `dataproc_start` - we use the google CreateDataProc operator in order to start a dataproc cluster that will be used to run pyspark scripts.

dataproc*start* >> [*dataproc*ingest*one*, *dataproc*ingest*two*, *dataproc*ingest_three]

dataproc*ingest*one >> dataproc*create*clickout

[dataproc*create*clickout, dataproc*ingest*two, dataproc*ingest*three] >> end



Example

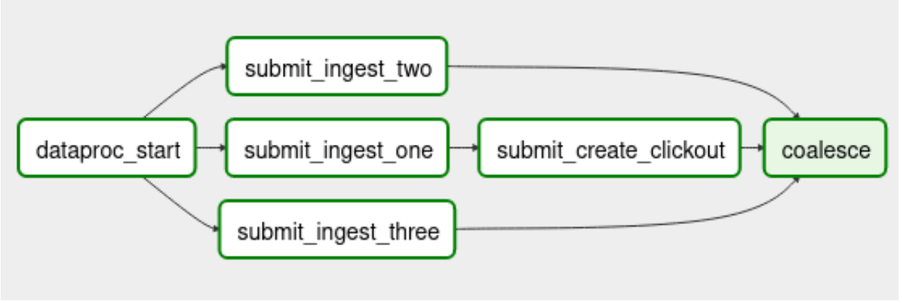
Let's say that you have to create an ETL process entirely done in the cloud, and we define the following tasks:

- `dataproc_start` - we use the google CreateDataProc operator in order to start a dataproc cluster that will be used to run pyspark scripts.
- `submit_ingest_one` - next up we create a data processing step to submit a pyspark job that will ingest data from somewhere, and will also be processed further. However, we want to make it so that if this fails, then whatever comes next fails as well.

dataproc*start* >> [*dataproc*ingest*one*, *dataproc*ingest*two*, *dataproc*ingest_three]

dataproc*ingest*one >> dataproc*create*clickout

[dataproc*create*clickout, dataproc*ingest*two, dataproc*ingest*three] >> end



Example

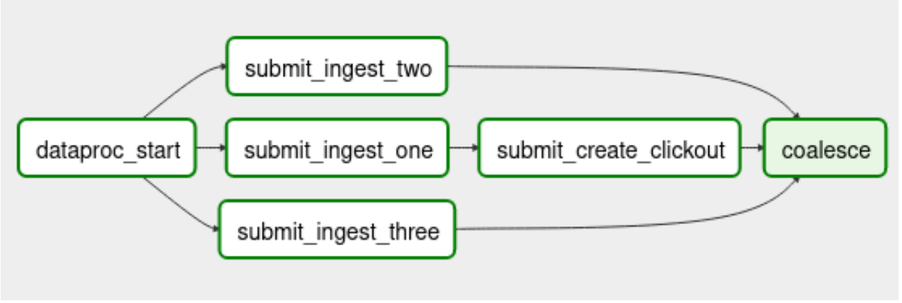
Let's say that you have to create an ETL process entirely done in the cloud, and we define the following tasks:

- `dataproc_start` - we use the google CreateDataProc operator in order to start a dataproc cluster that will be used to run pyspark scripts.
- `submit_ingest_one` - next up we create a data processing step to submit a pyspark job that will ingest data from somewhere, and will also be processed further. However, we want to make it so that if this fails, then whatever comes next fails as well.
- `submit_ingest_two, submit_ingest_three` - the same as `submit_ingest_one` except no other process depends on it

dataproc*start* >> [*dataproc*ingest*one*, *dataproc*ingest*two*, *dataproc*ingest_three]

dataproc*ingest*one >> dataproc*create*clickout

[dataproc*create*clickout, dataproc*ingest*two, dataproc*ingest*three] >> end



Example

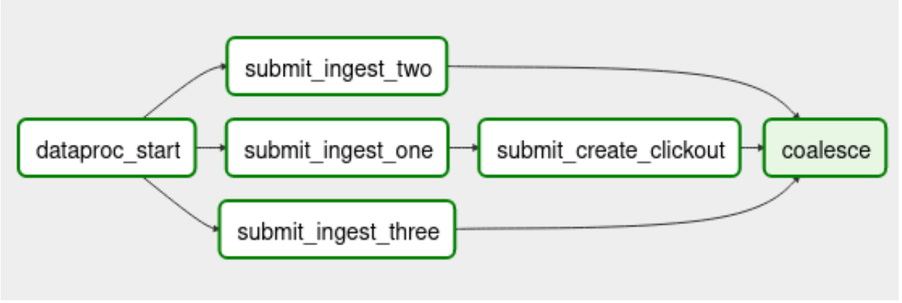
Let's say that you have to create an ETL process entirely done in the cloud, and we define the following tasks:

- `dataproc_start` - we use the google CreateDataProc operator in order to start a dataproc cluster that will be used to run pyspark scripts.
- `submit_ingest_one` - next up we create a data processing step to submit a pyspark job that will ingest data from somewhere, and will also be processed further. However, we want to make it so that if this fails, then whatever comes next fails as well.
- `submit_ingest_two, submit_ingest_three` - the same as `submit_ingest_one` except no other process depends on it
- `coalesce` - after all tasks have finished their execution, we would like to turn off the cluster.

dataproc*start* >> [*dataproc*ingest*one*, *dataproc*ingest*two*, *dataproc*ingest_three]

dataproc*ingest*one >> dataproc*create*clickout

[dataproc*create*clickout, dataproc*ingest*two, dataproc*ingest*three] >> end



Example

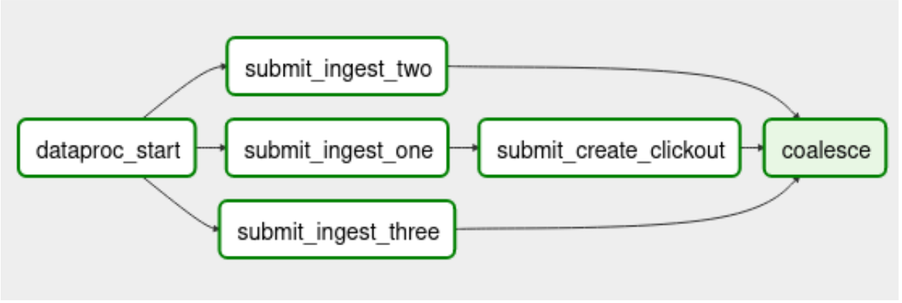
Let's say that you have to create an ETL process entirely done in the cloud, and we define the following tasks:

- `dataproc_start` - we use the google CreateDataProc operator in order to start a dataproc cluster that will be used to run pyspark scripts.
- `submit_ingest_one` - next up we create a data processing step to submit a pyspark job that will ingest data from somewhere, and will also be processed further. However, we want to make it so that if this fails, then whatever comes next fails as well.
- `submit_ingest_two, submit_ingest_three` - the same as `submit_ingest_one` except no other process depends on it
- `coalesce` - after all tasks have finished their execution, we would like to turn off the cluster.

dataproc*start* >> [*dataproc*ingest*one*, *dataproc*ingest*two*, *dataproc*ingest_three]

dataproc*ingest*one >> dataproc*create*clickout

[dataproc*create*clickout, dataproc*ingest*two, dataproc*ingest*three] >> end



Example

Let's say that you have to create an ETL process entirely done in the cloud, and we define the following tasks:

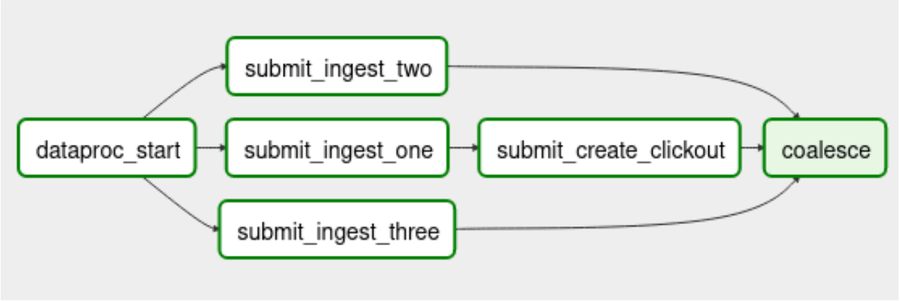
- `dataproc_start` - we use the google CreateDataProc operator in order to start a dataproc cluster that will be used to run pyspark scripts.
- `submit_ingest_one` - next up we create a data processing step to submit a pyspark job that will ingest data from somewhere, and will also be processed further. However, we want to make it so that if this fails, then whatever comes next fails as well.
- `submit_ingest_two, submit_ingest_three` - the same as `submit_ingest_one` except no other process depends on it
- `coalesce` - after all tasks have finished their execution, we would like to turn off the cluster.

- We could define that with the following edge configuration:

dataproc*start* >> *[dataproc*ingest*one*, *dataproc*ingest*two*, *dataproc*ingest_three]

dataproc*ingest*one >> dataproc*create*clickout

[dataproc*create*clickout, dataproc*ingest*two, dataproc*ingest*three] >> end



Trigger Rules

Trigger rules are, in airflow-speech, a name for how should the node at the end of the edge behave when some exit status comes out of its parents.

Trigger rules can be added to any operator with the argument `trigger_rule`, and there are quite a couple for you to choose from, out of which the following are easily exemplified:

Trigger Rules

Trigger rules are, in airflow-speech, a name for how should the node at the end of the edge behave when some exit status comes out of its parents.

Trigger rules can be added to any operator with the argument `trigger_rule`, and there are quite a couple for you to choose from, out of which the following are easily exemplified:

- `all_success` - use this when you want everything to work in the most obvious way as possible, and whatever you are doing happens to be very linear.

Trigger Rules

Trigger rules are, in airflow-speech, a name for how should the node at the end of the edge behave when some exit status comes out of its parents.

Trigger rules can be added to any operator with the argument `trigger_rule`, and there are quite a couple for you to choose from, out of which the following are easily exemplified:

- `all_success` - use this when you want everything to work in the most obvious way as possible, and whatever you are doing happens to be very linear.
- `all_done` - this is very often used whenever there's a cluster start or end job. For instance, let's say that irrespective of the parent tasks status, given that they have finished their execution, you would like this task to run anyways. It's commonly used to shut down a cluster.

Trigger Rules

Trigger rules are, in airflow-speech, a name for how should the node at the end of the edge behave when some exit status comes out of its parents.

Trigger rules can be added to any operator with the argument `trigger_rule`, and there are quite a couple for you to choose from, out of which the following are easily exemplified:

- `all_success` - use this when you want everything to work in the most obvious way as possible, and whatever you are doing happens to be very linear.
- `all_done` - this is very often used whenever there's a cluster start or end job. For instance, let's say that irrespective of the parent tasks status, given that they have finished their execution, you would like this task to run anyways. It's commonly used to shut down a cluster.
- `none_failed` - is most often than not paired with the branching operator. Maybe you will need to send the data through a different path in the dag and thus it is acceptable to skip a chunk of it. If you want your dag execution to continue, you have to make it so that the parent tasks were either skipped or successfully completed.

Trigger Rules

Trigger rules are, in airflow-speech, a name for how should the node at the end of the edge behave when some exit status comes out of its parents.

Trigger rules can be added to any operator with the argument `trigger_rule`, and there are quite a couple for you to choose from, out of which the following are easily exemplified:

- `all_success` - use this when you want everything to work in the most obvious way as possible, and whatever you are doing happens to be very linear.
- `all_done` - this is very often used whenever there's a cluster start or end job. For instance, let's say that irrespective of the parent tasks status, given that they have finished their execution, you would like this task to run anyways. It's commonly used to shut down a cluster.
- `none_failed` - is most often than not paired with the branching operator. Maybe you will need to send the data through a different path in the dag and thus it is acceptable to skip a chunk of it. If you want your dag execution to continue, you have to make it so that the parent tasks were either skipped or successfully completed.
- `all_failed` - ~~pair this one up with a callback so you can notify yourself of getting fired~~