Cascading

?

- Query API and query Planner
- used for defining, sharing & execute data processing workflows on cluster
- simplifies the complexity with hadoop application development

github.com/BlackRabbitt/LevelUpCascading.git

level 0: know the basics

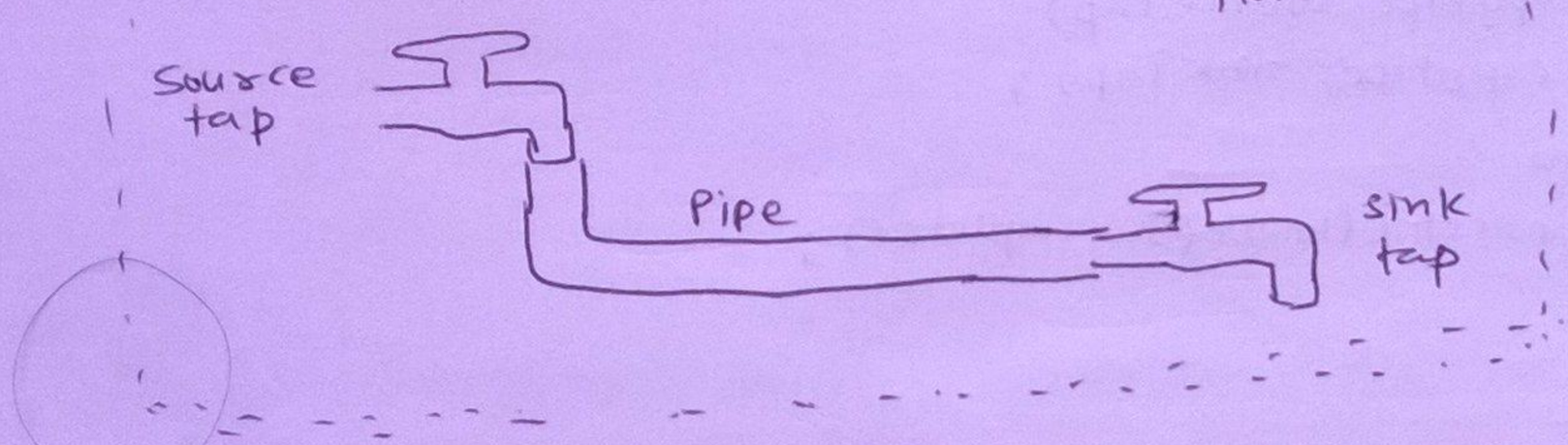


fig: copy file from source tap to sink tap

Source tap & sink tap

→ Source tap: All input data comes from &
Sink Tap: All output data reads to.
→ cascading.tap.Tap instance.
→ Tap represents a resources like a data file on the local file system or on a Hadoop distributed file system.
→ Taps associated with scheme.

Tap: where the data is & how to get it ?
- LFS: refer local files
- DFS: refer HDFS files
- HFS: uses the current HDFS

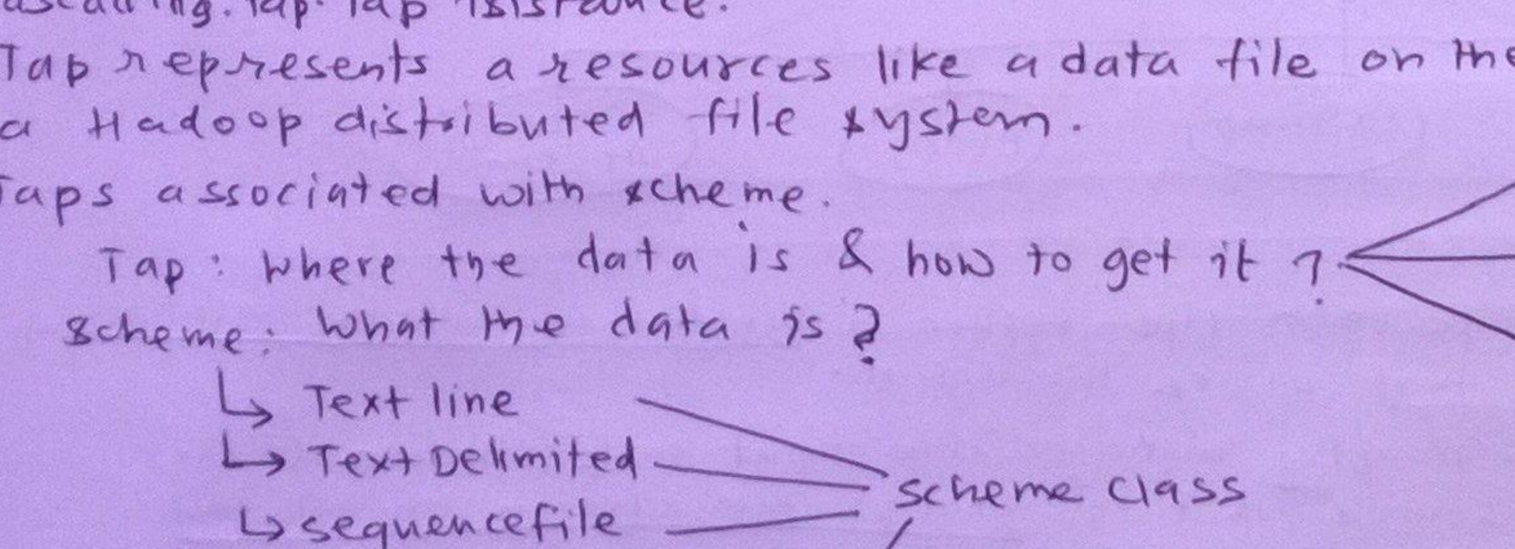
(Tap classes)

Scheme: What the data is ?
- Text line
- Text Delimited
- sequencefile
- Writable sequencefile

(scheme class)

① Text line :- reads and write raw text files & returns Tuples with two fields names by default. "offset" & "line".
② Text Delimited :- reads and write character delimited files (.csv, etc).

Tuple, field, record

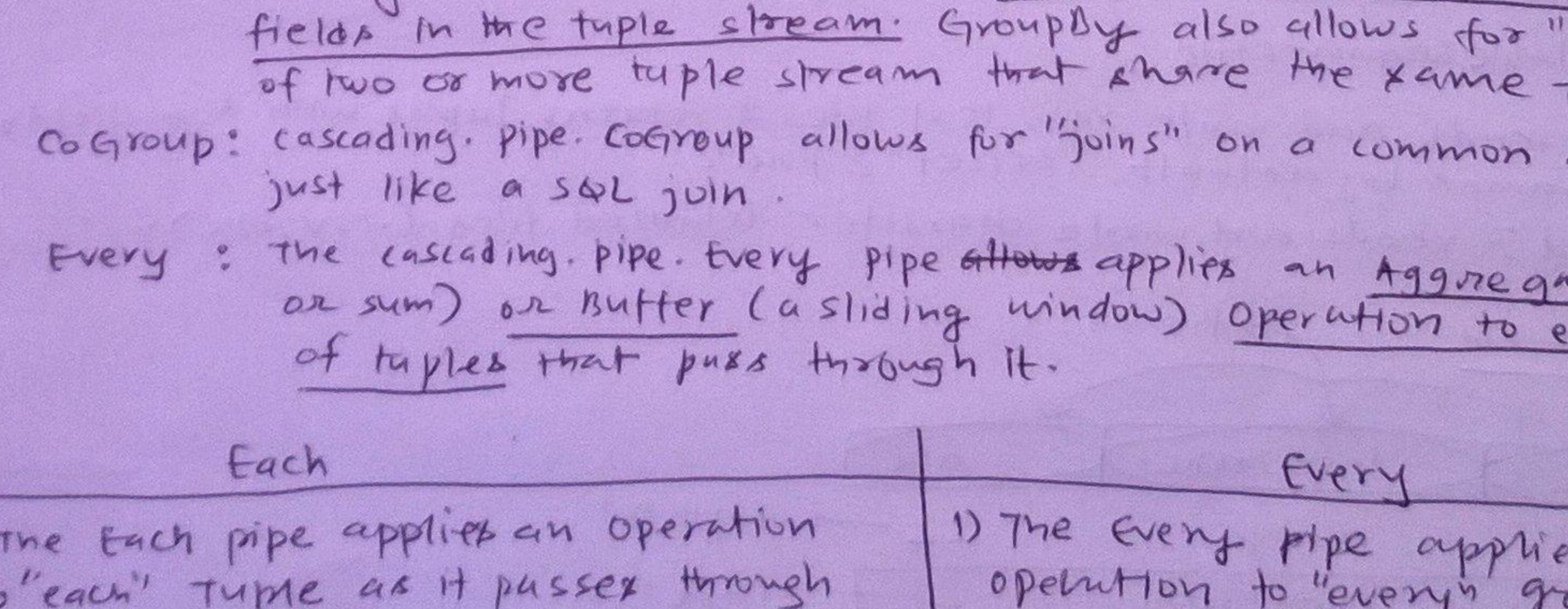


flow is executed to push the incoming source data through the assembly into one or more sinks.

```
// copy files from source tap to sink tap
// First create a source tap
String inPath = path_to_your_file ;
Tap sourceTap = new Hfs( new TextDelimited (true, "\t"), inPath);
// create a sink tap
string outPath = path_to_where_you_want_to_copy_files;
Tap sinkTap   = new Hfs( new TextDelimited (true, "\t"), outPath);
// create a pipe.
Pipe copyPipe = new Pipe("copy");
// connect tap & pipe to create a flow.
FlowDef flowDef = FlowDef.flowDef()
    .addSource( copyPipe, sourceTap)
    .addTailSink( copyPipe, sinkTap);
// complete the flow
flowConnector.connect(flowDef).complete();
```

Level 1 : Pipe Assemblies

Pipe assemblies define what work should be done against a tuple stream, where during runtime tuple streams are ~~use~~ read from tap source & are written to tap sinks.



Each : The cascading.pipe.Each pipe applies a function or filters operation to each Tuple that passes through it.

GroupBy : cascading.pipe.GroupBy manages one input tuple stream & does exactly as it sounds, that is, groups the stream on selected fields in the tuple stream. GroupBy also allows for "merging" of two or more tuple stream that share the same field names.

CoGroup : cascading.pipe.CoGroup allows for "joins" on a common set of values, just like a SQL join.

Every : The cascading.pipe.Every pipe ~~allows~~ applies an Aggregator (like count or sum) or Buffer (a sliding window) operation to every group of tuples that pass through it.

Each	Every
1) The Each pipe applies an operation to "each" Tuple as it passes through the pipe assembly.	1) The Every pipe applies an operation to "every" group of Tuples as they pass through the pipe assembly, on the tail end of a GroupBy or CoGroup pipe.
2) The Each pipe may only apply Functions & Filters to the tuple stream as these operations may only operate on one Tuple at a time.	2) The Every pipes may only apply Aggregators & Buffers to the tuple stream as these operations may only operate on groups of tuples, one grouping at a time.

GroupBy accepts one or more tuple streams. If two or more, they must all have the same field names (this is also called a merge).

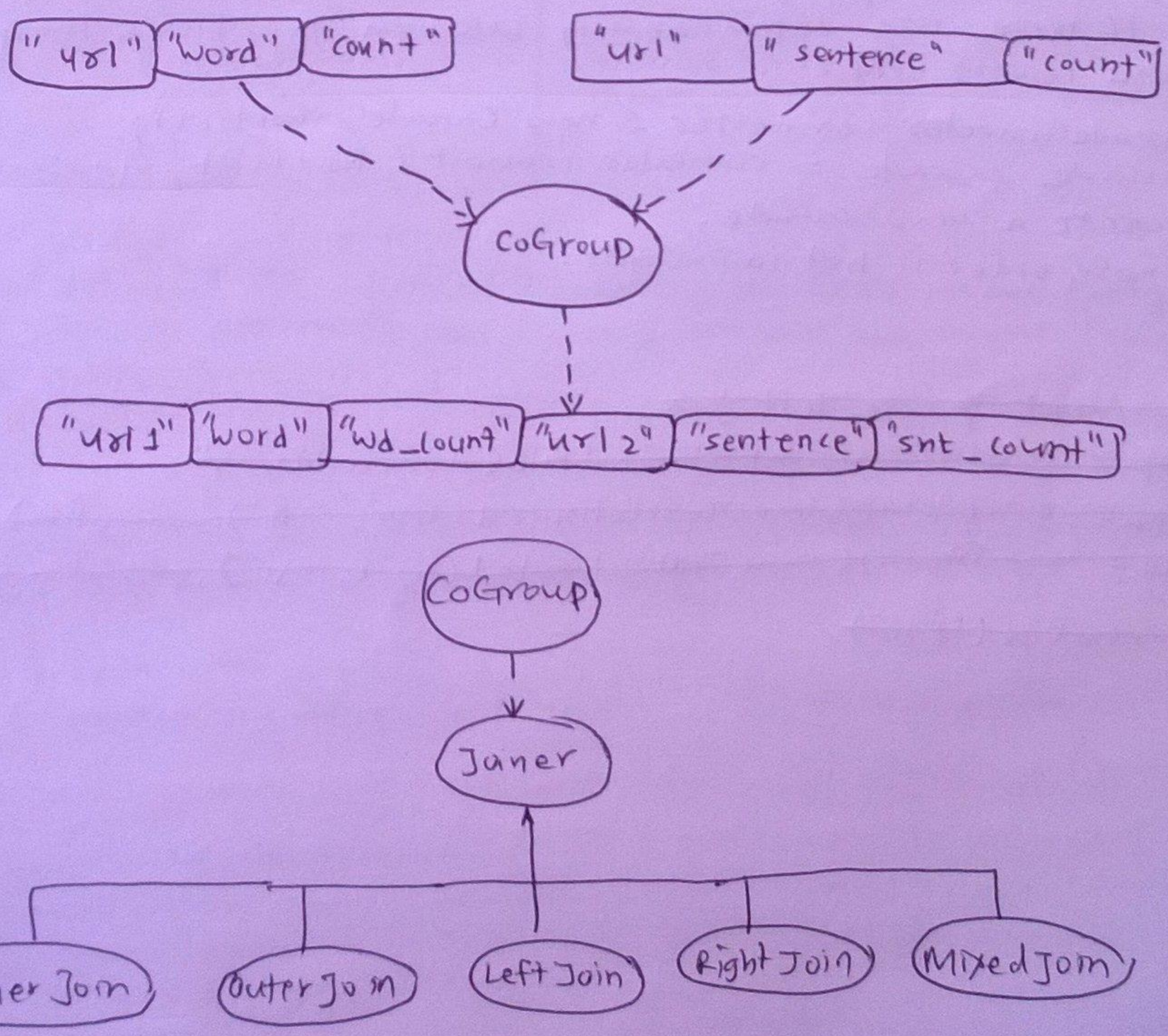
eg. `Pipe groupByPipe = new GroupBy( assembly, new Fields("a","b"));`

The example above simply creates a new tuple stream where Tuples with the same value in "a" & "b" can be processed as a set by an Aggregator or Buffer operation. The resulting streams of tuples will be sorted by the values in "a" & "b".

CoGroup accepts two or more tuple streams and does not require any common field names. The grouping fields must be provided for each tuple stream.

```
eg. Fields lhsF = new Fields("field A", "field B");
    Fields rhsF = new Fields("field C", "field D");
    Pipe join = new CoGroup(lhs, lhsF, rhs, rhsF, new InnerJoin());
```

This example joins two streams ("lhs" & "rhs") on common values. Note that common field names are not ~~allowed~~ required here. Actually, if there were any common field names, the cascading planner would throw an error as a duplicate field names are not allowed.



Eg.

LHS = [0,a] [1,b] [2,c] RHS = [0,A] [2,C] [3,D]

~~Outer~~ Join ⇐ [0,a,0,A] [1,b,null,null] [2,c,2,C] [null,null,3,D]

~~Inner~~ Join ⇐ [0,a,0,A] [2,c,2,C]

Left Join ⇐ [0,a,0,A] [1,b,null,null] [2,c,2,C]

Right Join ⇐ [0,a,0,A] [2,c,2,C] [null,null,3,D]

Flows

→ When pipe assemblies are bound to source & sink taps, a Flow is created.
→ Think of a Flow as a data processing workflow that reads data from sources, process the data as defined by the pipe assembly & writes data to the sink.

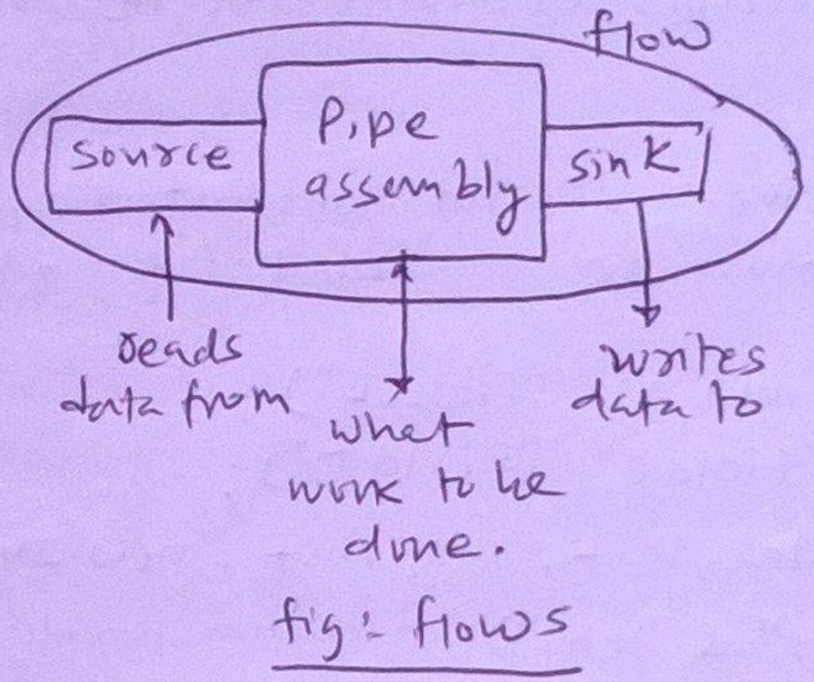


fig:- flows

```
Flow flow = new FlowConnector().
                connect("flow-name",
                        source,
                        sink,
                        pipe);
```

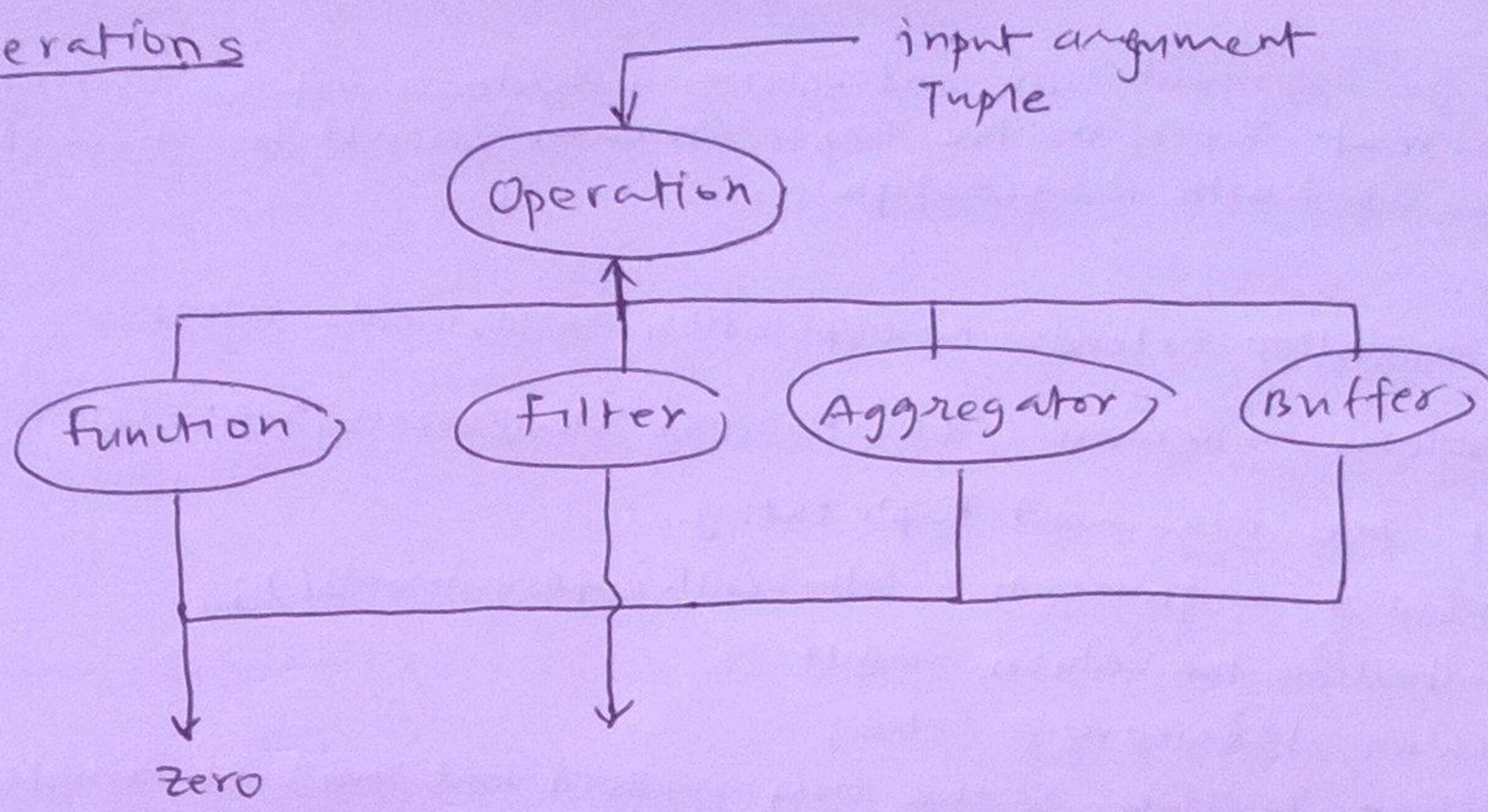
Cascade

A cascade allows multiple Flow instances to be executed as a single logical unit. If there are dependencies between the Flows, they will be executed in the correct order.

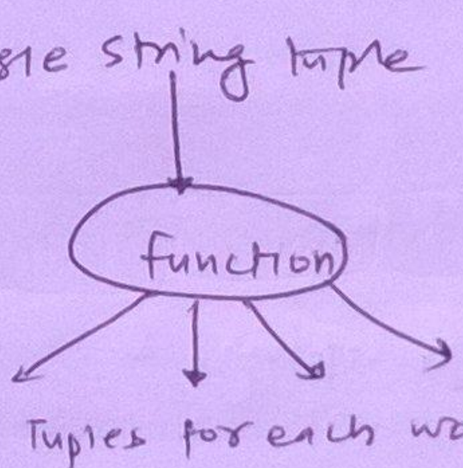
```
CascadeConnector connector = new CascadeConnector();
Cascade cascade = connector.connect( FlowFirst, FlowSecond, FlowThird);
// create a new cascade.
// note order is not important.
```

~~// Join equivalent to SQL left join.~~
~~Tap inputTap = getPlatform().getDelimitedFile(inputFields, ",",~~
~~Tap sourceTap = new FileTap(new TextDelimited(true, "#"), sourcePath);~~
~~Tap sinkTap = new FileTap(new TextDelimited(true, "|"), sinkPath, sinkMode.REPLACE);~~
~~Pipe pipe = new Pipe("pipe")~~

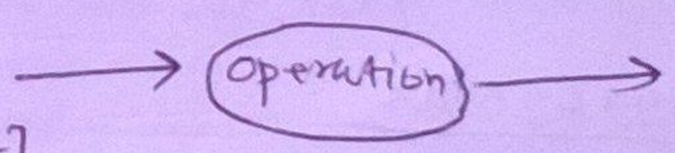
Level 2 : Operations



* eg



During runtime,



→ All operation must be wrapped by either an Each or an Every pipe instance. The pipe is responsible for parsing in an argument Tuple and accepting the result Tuple.

1. Function

→ expects a single arguments Tuple, and may return zero or more result tuples.
→ only used with a Each pipe which may follow any other pipe type.

```
// creating a custom function.
public class someFunction extends BaseOperation implements Function
{   public void operate( FlowProcess flowProcess, FunctionCall functionCall)
    {
        // get the arguments TupleEntry
        TupleEntry arguments = functionCall.getArguments();
        //create a Tuple to hold our result values
        Tuple result = new Tuple();
        //insert some values into the result Tuple

        //return the result Tuple
        functionCall.getOutputCollector().add(result);
    }
}
```

2. Filter

→ Expects a single argument Tuple and returns a boolean value stating whether or not the current tuple in the tuple stream should be discarded.
→ may only be used with a Each pipe.

```
//custom filter.
public class SomeFilter extends BaseOperation implements Filter.
{
    public boolean isRemove ( FlowProcess FlowProcess, Filter Call filterCall)
    {
        //get the argument TupleEntry.
        TupleEntry arguments = filterCall.getArguments();
        // initialize the return result
        boolean isRemove = false;
        // Filter interface is the only method that must be implemented.
        // test the argument values and set isRemove accordingly
        return isRemove;
    }
}
```

3. Aggregator

→ Expects set of argument Tuples in the same grouping, & may return zero or more result Tuples.
→ may only be used with an Every pipe, and it may follow a GroupBy, CoGroup, or another Every pipe type.

```
//custom Aggregator
public class SomeAggregator extends BaseOperation <someAggregator.Context>
Implements Aggregator <someAggregator.Context>
{   public static class Context
    {
        object value;
    }
    public void start (FlowProcess flowProcess,
                       AggregatorCall <Context> aggregatorCall)
    {   //get the group values for the current grouping.
        TupleEntry group = aggregatorCall.getGroup();
        //create a new custom context object
        Context context = new Context();
        //optionally, populate the context object
        // set the context object
        aggregatorCall.setContext (context);
    }
public void aggregate (FlowProcess flowProcess,
                       AggregatorCall <context> aggregatorCall)
    {
        //set the current argument values
        TupleEntry arguments = aggregatorCall.getArguments();
        //get the context for this grouping.
        Context context = aggregatorCall.getContext();
        //update the context object.
    }
```

```
public void complete( FlowProcess flowProcess,
                      AggregatorCall <context> aggregatorCall )
  {
    Context context = aggregatorCall.getContext();
    // create a Tuple to hold our result values
    Tuple result = new Tuple();
    // insert some values into the result Tuple based on the context
     aggregateCall.getOutputCollector().add(result);
    }
}
```

4. Buffer

→ expects set of arguments Tuples in the same grouping, and may return zero or more tuples.

→ The Buffer is very similar to an Aggregator except it receives the current Grouping ~~tuple~~ Tuple and an iterator of all the arguments it excepts for every value Tuple in the current grouping, all on the same method call.

→ This is very similar to the typical Reducer interface, and is best used for operations that need greater visibility to the previous & next elements in the stream.

→ is used with an every pipe, and it may be only follow a GroupBy & CoGroup pipe type.

```
// custom buffer
public class SomeBuffer extends BaseOperation implements Buffer
  { public void operate( FlowProcess flowProcess, BufferCall bufferCall)
  {
    // get the group values for the current grouping
    TupleEntry group = bufferCall.getGroup();
    // get all the current argument values for this grouping
    Iterator<TupleEntry> arguments = bufferCall.getArgumentsIterator();
    // create a tuple to hold our result values
    Tuple result = new Tuple();
    while ( arguments.hasNext())
      {
        TupleEntry argument = arguments.next();
        // insert some values into the result Tuple based on
          the arguments
      }
    // return the result Tuple
    bufferCall.getOutputCollector().add(result);
  }
}
```

Level 3 Built in Operations

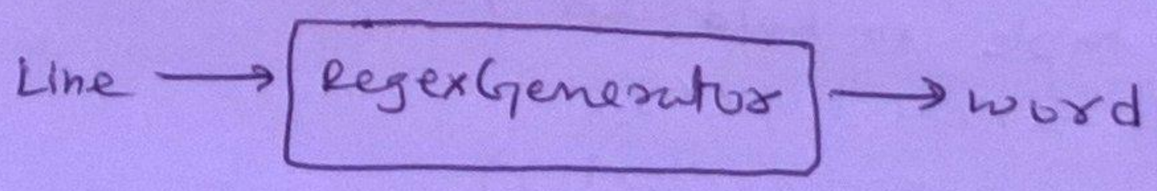
1. Identity function
   → Is used to "shape" a tuple stream.
   → Discard unused fields

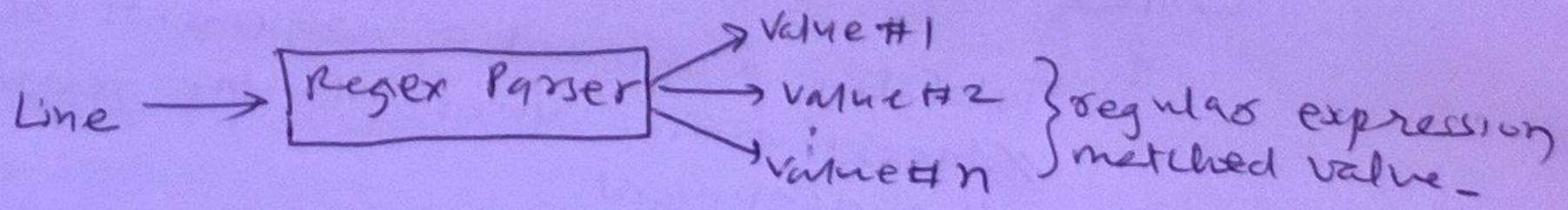
```
// incoming → "ip", "time", "method", "event"
Pipe = new Each (pipe, new Fields ("ip", "method"), new Identity())
//outgoing → "ip", "method"
```

   → Or, Rename all fields

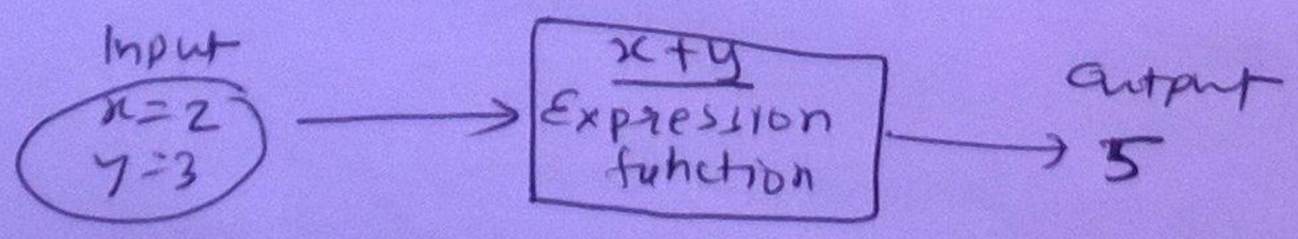
```
// incoming → "method"
Identity identity = new Identity (new Fields ("methods#2"));
Pipe = new Each (pipe, new Fields("method"), identity);
// outgoing => "methods#2"
```

2. Regular Expression Operations.
   ⓐ Regex Splitter
   → will split an argument value by a regex pattern ~~string~~ string.
   ⓑ Regex Parser
   → is used to extract a regular expression matched value from an incoming argument value.
   ⓒ Regex Replace
   → is used to replace a regex matched value with a replacement value. It may be used in a "replace all" or "replace first" mode.
   ⓓ Regex filter
   → apply a regular expression pattern string against every input Tuple value and filter the tuple stream accordingly. By default, the Tuples that match the given pattern are kept & tuples that do not match are filtered out.
   ⓔ Regex Generator
   → function will emit a new Tuple for every matched regular expression group,





3. Java Expression Operations
   ⓐ Expression function.
   → function dynamically resolves a given expression using argument Tuple values as inputs to the fields specified in the expression.



   ⓑ ExpressionFilter
   → Dynamically resolves a given expression using argument Tuple values as inputs to the fields specified in the expression. Any Tuple that returns true for the given expression will be removed from the stream

$x = [3, 4, 7, 9] \longrightarrow [x \geq 5] \longrightarrow x = [3, 4]$ [$x \geq 5$ is removed]