eHive Workshop

part 3: How to create pipelines (configuration files)

Instructors:
Leo Gordon and Brandon Walts

Before we begin

- Have a couple of terminal windows open, and be sure you are in the ehive-course directory

```
$ cd $HOME/ehive-course
```

- Double-click on "Start guiHive server" to be sure it is running
  - Note, there is no immediate feedback when you do this.

A quick note on Perl package names and paths for Perl non-experts

- Perl package names are hierarchical, with the names separated::by::double::colons::like::this.
- Package names are designated with the keyword 'package', imported using 'use' (or 'require' but that's less common) and set as a base using 'use base'
  - e.g.
    ```
    package Awesome::Alignment::Parser;
    use base ('Awesome::Generic::Parser');
    ```
- The package name has to match the filename and the path to the file.
  - So the path to class `Awesome::Alignment::Parser` has to be `Awesome/Alignment/Parser.pm`
- Perl starts looking for these paths:
  - in the current directory
  - in the directories in `$PERL5LIB`
  - in a list called `@INC`
    - You can modify `@INC` several ways, one is to call perl with `-I`

Modularity of pipelines. PipeConfigs & Runnables

- A Hive pipeline is defined in a PipeConfig file.
  - These have a naming convention `NameOfPipeline_conf.pm`
- A PipeConfig contains:
  - Meta-information about the pipeline (such as default parameters)
  - The pipeline structure itself. In a nutshell, the structure is a list of the Runnables that will do the work, how they are linked to each other, and how data flows between them.

- Many tasks can be solved by using “universal” Runnables provided by the Hive (SystemCmd, SqlCmd, JobFactory, Dummy), but sometimes you have to write your own application-specific Runnables.

- We shall learn to use universal Runnables before making our own

- Hive’s “universal” Runnables live here:

  `$ENSEMBL_REPO_ROOT_DIR/ensembl-hive/modules/Bio/EnsEMBL/Hive/RunnableDB/`

- Hive’s PipeConfig files live here:

  `$ENSEMBL_REPO_ROOT_DIR/ensembl-hive/modules/Bio/EnsEMBL/Hive/PipeConfig/`

- There are also useful Runnable and PipeConfigs here:

  `$ENSEMBL_REPO_ROOT_DIR/ensembl-hive/modules/Bio/EnsEMBL/Hive/Examples/`

The simplest pipeline : AnyCommands_conf.pm

- This is the smallest PipeConfig possible:
- Under the examples directory:

```
$ENSEMBL_REPO_ROOT_DIR/ensembl-hive/modules/Bio/EnsEMBL/Hive/Examples
```

- In the SystemCmd examples:

```
SystemCmd/PipeConfig/AnyCommands_conf.pm
```

```
use base ('Bio::EnsEMBL::Hive::PipeConfig::HiveGeneric_conf');      #  or subclass


sub pipeline_analyses {
    return [
        {   -logic_name    => 'perform_cmd',
            -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
        },
    ];
}
```

Bio::EnsEMBL::Hive::RunnableDB::SystemCmd

```
use base ('Bio::EnsEMBL::Hive::PipeConfig::HiveGeneric_conf');      #  or subclass


sub pipeline_analyses {
    return [
        {   -logic_name    => 'perform_cmd',
            -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
        },
    ];
}
```

A simple Runnable:

- You can pass it a parameter called "cmd" - the value is a command like you would type at a command prompt.
- It runs that command in a shell, just like if you'd typed it at a command prompt ("does exactly what it says on the tin!")

The simplest pipeline : AnyCommands_conf.pm

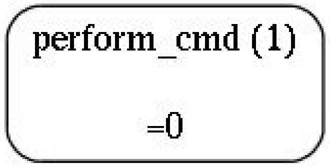
- Exercise - let's initialize this pipeline and work with it for a bit

```
# it can be handy to store the pipeline url in an environment variable
$ export EHIVE_URL=mysql://ensrw:ensrw_password@localhost/my_pipeline_db

$ init_pipeline.pl \
Bio::EnsEMBL::Hive::Examples::SystemCmd::PipeConfig::AnyCommands_conf \
-pipeline_url $EHIVE_URL


$ generate_graph.pl -url $EHIVE_URL -out any_c_empty.png
```

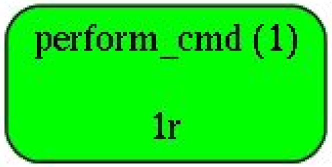
or open guiHive http://127.0.0.1:8080/
and connect to `mysql://ensrw:ensrw_password@localhost/anycommands_hive_db`



Seeding and running AnyCommands_conf.pm

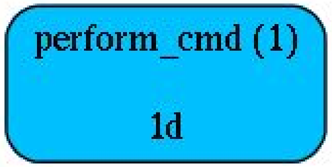
- No jobs yet, so we need to seed one using seed_pipeline.pl:

```
$ seed_pipeline.pl -url $EHIVE_URL -logic_name perform_cmd \
    -input_id '{"cmd" => "echo Hello, world"}'
```



- now we can run a worker:

```
$ runWorker.pl -url $EHIVE_URL
```



Analysis-wide parameters and substitution

- We can define values for old parameters and create new ones:
- Open AnyCommands_conf.pm and make the following change:

```
sub pipeline_analyses {
    return [
        {   -logic_name    => 'perform_cmd',
            -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
            -parameters => {
                 "cmd" => "gzip #filename# ",
            },
        },
    ];
}
```

- Two things going on here:
  - We're setting a default parameter for all jobs of an analysis
  - We're adding a new substitutable parameter using the #parameter_name# syntax

Analysis-wide parameters and substitution

- Exercise:
  - Save your changes to AnyCommands_conf.pm
  - Re-initialize your AnyCommands pipeline
  - Seed a few jobs to compress a file and run them (there should be some .fa files in the ehive-course directory you can compress)

```
$ seed_pipeline.pl -url $EHIVE_URL -logic_name perform_cmd \
     -input_id '{"filename" => "j_random_fasta_file.fa"}'
```

Working towards automation...

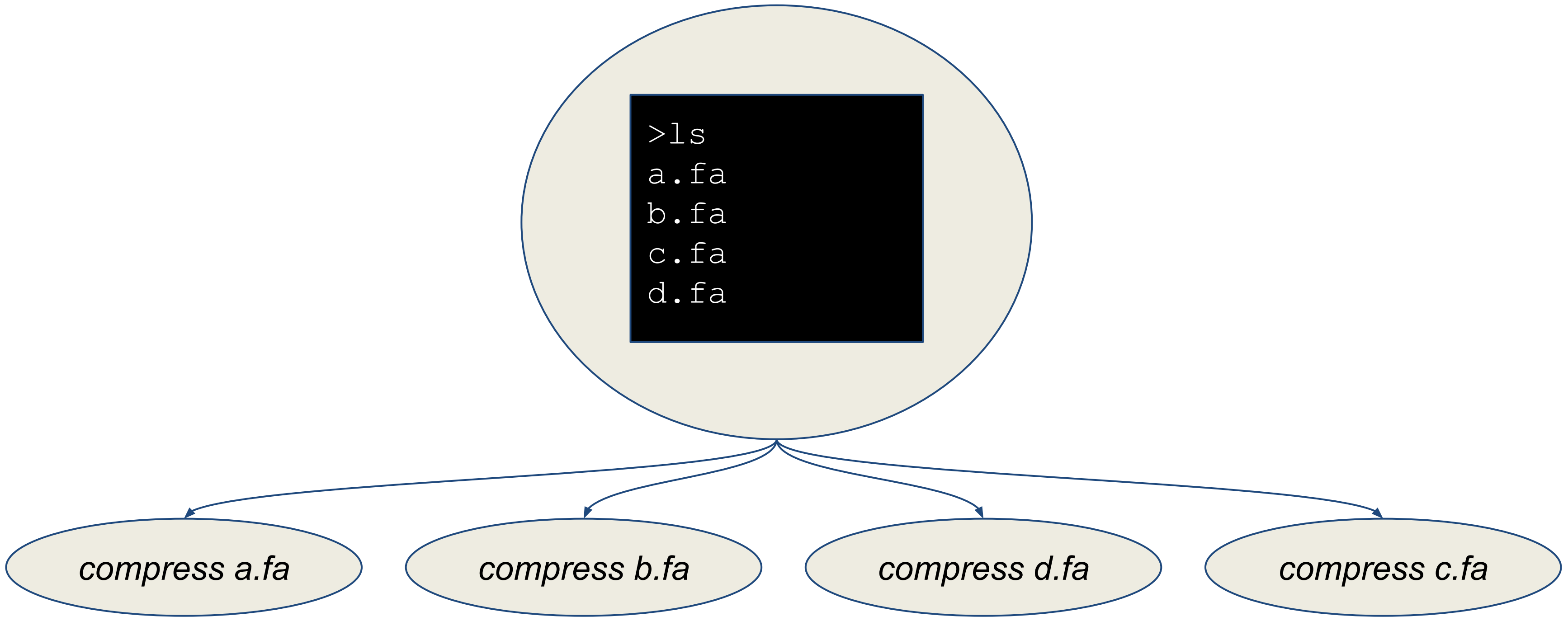
- Let's say we want to compress a lot of files.
- You could seed multiple jobs with a bash shell loop like this:

```
for filename in `find . -name '*.fa'` ; do
    seed_pipeline.pl -url $EHIVE_URL \
        -logic_name perform_cmd -input_id "{ 'filename' => '$filename' }";
done
```

- However, that's not very eHive-ish
- After all, one of eHive's key features is to manage multiple jobs like this for you.

Working towards automation...

- Some runnables can seed multiple jobs* when their jobs are run
- In general, we call a step in the pipeline that creates a fan of parallel jobs a "factory"
- A couple of factories are provided in eHive's standard runnable library
  - FastaFactory
  - JobFactory
- We'll work with JobFactory to improve our file compression pipeline. We'll use it to take a list of files and create a new compress a file job for each file in the list.



```
>ls
a.fa
b.fa
c.fa
d.fa
```

*More correctly, they can create multiple events on the same branch when their jobs run. Seeding a job is just one of many things an event can trigger.

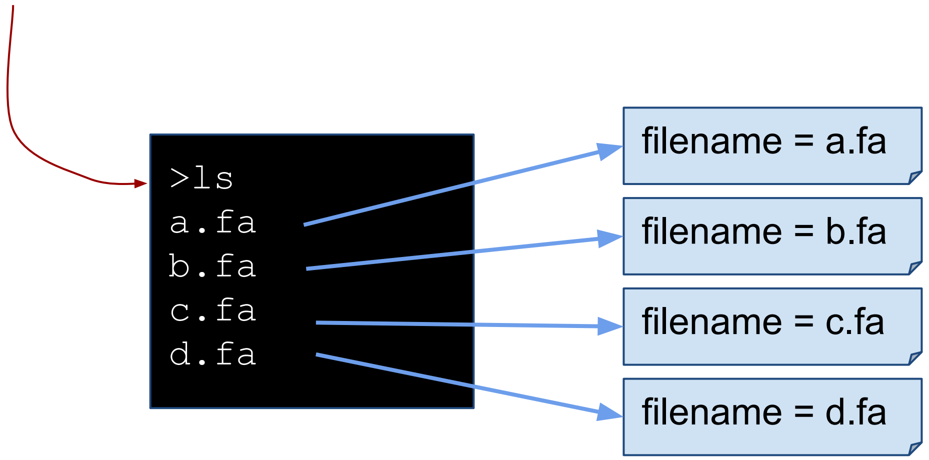
JobFactory

- Like SystemCmd, JobFactory runs a command (for the record, it can do other things, like run SQL against a database).
- Unlike SystemCmd, JobFactory captures the output of the command. It can package up the output and pass it along the pipeline, line by line.

Input parameters we'll use (for the full list see the module's documentation)

- inputcmd - the command JobFactory will run (similar to what we saw previously with SystemCmd's "cmd")
- column_names - this is the name(s) we want to call the parameter(s) that contain the output.
  - For ls, typically there is only one column of output, so you'd pass one column name to capture it.

```
… -input_id '{"inputcmd" => " ls", "column_names" => ["filename"]}'
```



Factories and dataflow : CompressFiles_conf.pm

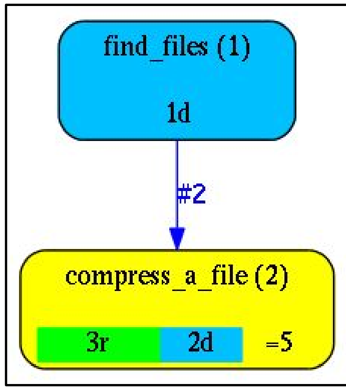
- Under `Examples/Factories/PipeConfig`
- Here, we're wiring two analyses together with -flow_into
- New first analysis using JobFactory, second analysis is familiar, using SystemCmd

```
sub pipeline_analyses {
    return [
        {   -logic_name => 'find_files',
            -module     => 'Bio::EnsEMBL::Hive::RunnableDB::JobFactory',
            -parameters => {
                'inputcmd'     => 'find #directory# -type f ',
                'column_names' => [ 'filename' ],
            },
            -flow_into => {
                2 => [ 'compress_a_file' ],
            },
        },
        {   -logic_name    => 'compress_a_file',
            -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
            -parameters    => {
                'cmd'       => 'gzip #filename#',
            },
            -analysis_capacity => 4,
        },
    ];
}
```

create a new higher-level parameter

what to call the output

where the output should go



Factories and dataflow : CompressFiles_conf.pm

Exercise - use CompressFiles_conf to compress all the files in a directory:

- Create a new directory in your ehive-course directory and put a few files in it - this is what the pipeline will compress
- Initialize the pipeline
  - The pipeconfig is `Bio::EnsEMBL::Hive::Examples::Factories::PipeConfig::CompressFiles_conf`
- Seed a job in that pipeline
  - Hint: you'll seed one 'find_files' job, which will need one parameter in the input_id hash.
- Look at the job table in the database, see the seeded job and its parameters
- Run one step of the pipeline with runWorker.pl
- Look at the pipeline in guiHive or at the database tables, notice several new jobs have been seeded
- Finish running the pipeline with beekeeper -loop -sleep 0.1
- Confirm that the files are compressed after the pipeline runs

Dataflow conventions

- Each Dataflow Event is a pair (branch_number, hash_of_parameters+).
- In our example a 'find_files' job that seeded with
  `{ 'directory' => 'fastas' }`
  would generate the following Dataflow Events:

```
#2, { 'filename' => 'fastas/first.fa' }
#2, { 'filename' => 'fastas/second.fa'}
. . .
#2, { 'filename' => 'fastas/last.fa' }
#1, { 'directory' => 'fastas' }
```

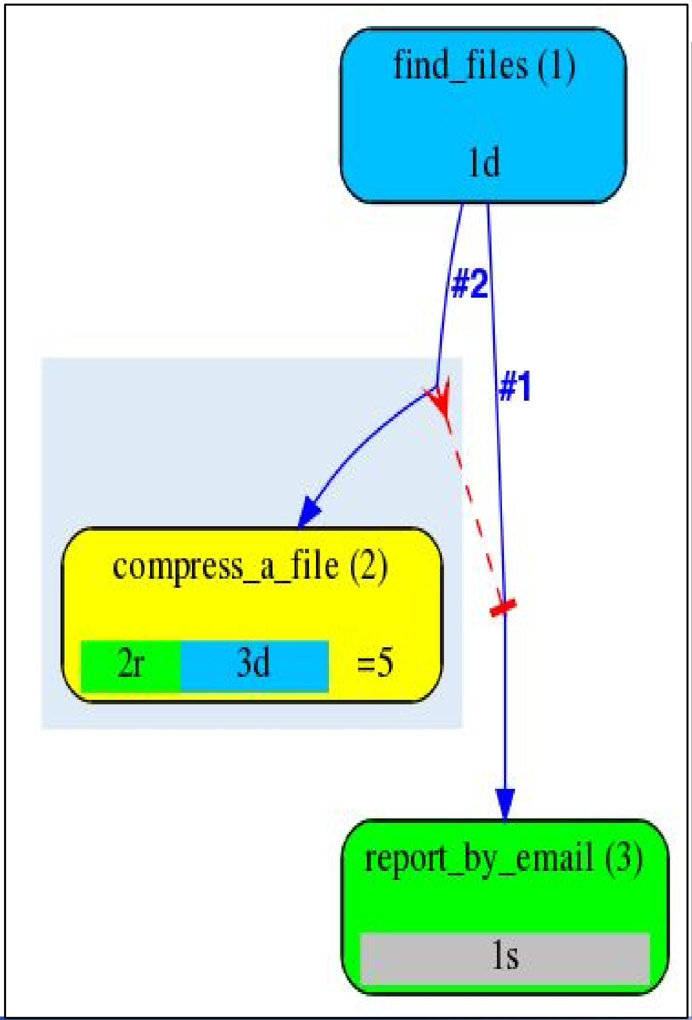
The "fan"

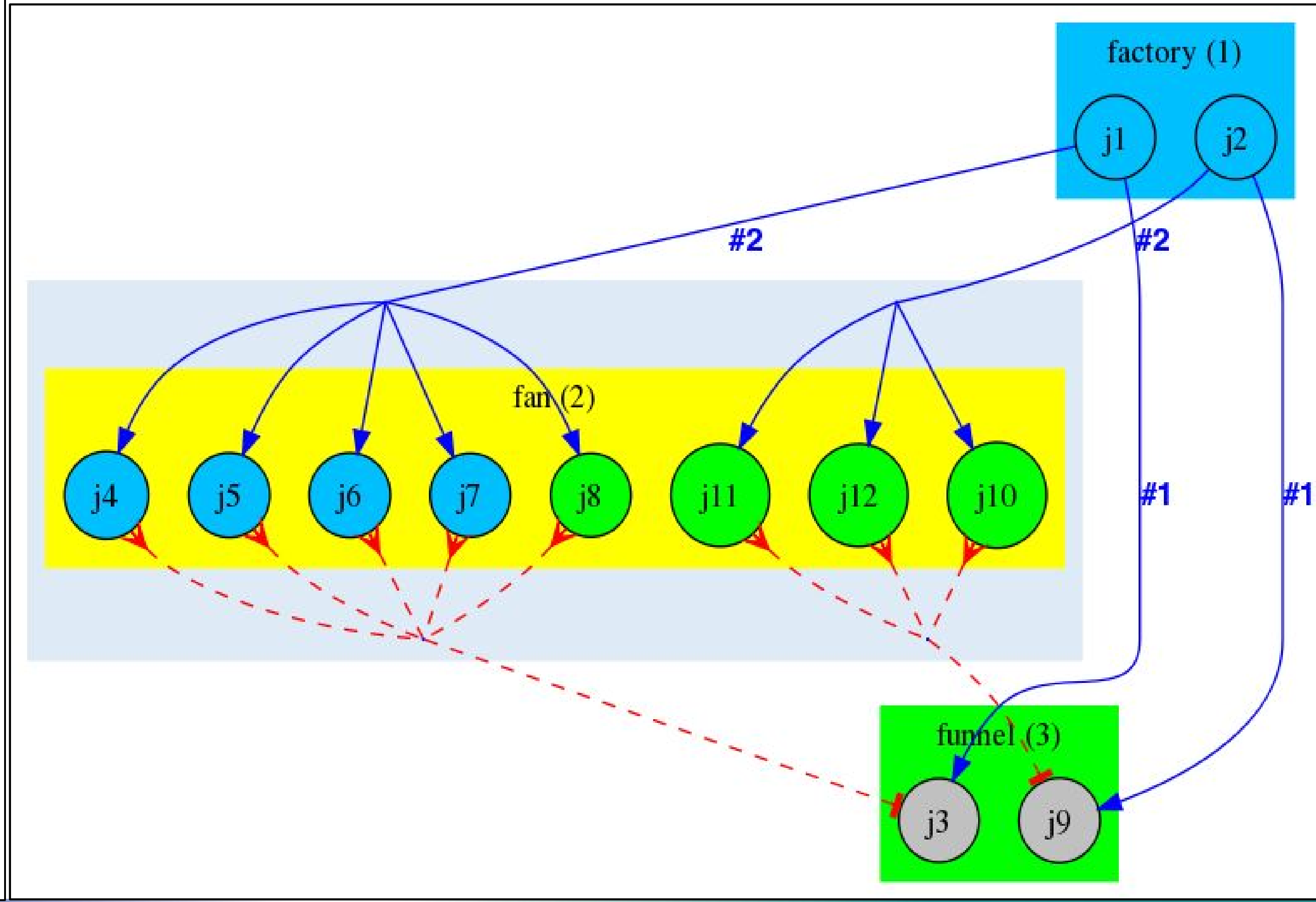
"autoflow" event, helps to bind analyses consecutively

- There are "reserved” branch numbers that have their own meaning (similar to UNIX file descriptors):
  - 1 is almost always present, it is the “continuation” after Job is ‘DONE’
  - 2 is used by many Factory Runnables to emit the “fan” (of Jobs, etc)
  - -1 : “postmortem dataflow after MEMLIMIT” on LSF
  - -2 : “postmortem dataflow after RUNLIMIT” on LSF
  - 3, 4, 5... : unreserved, can be used for anything
- Each Runnable has its own set of branch_numbers that it may emit Dataflow Events into.
- Check its documentation or code to make sure you get dataflow events on the branch you're wiring.
  - What happens if there are no events on that branch?

Regaining single thread of control: semaphored dataflow

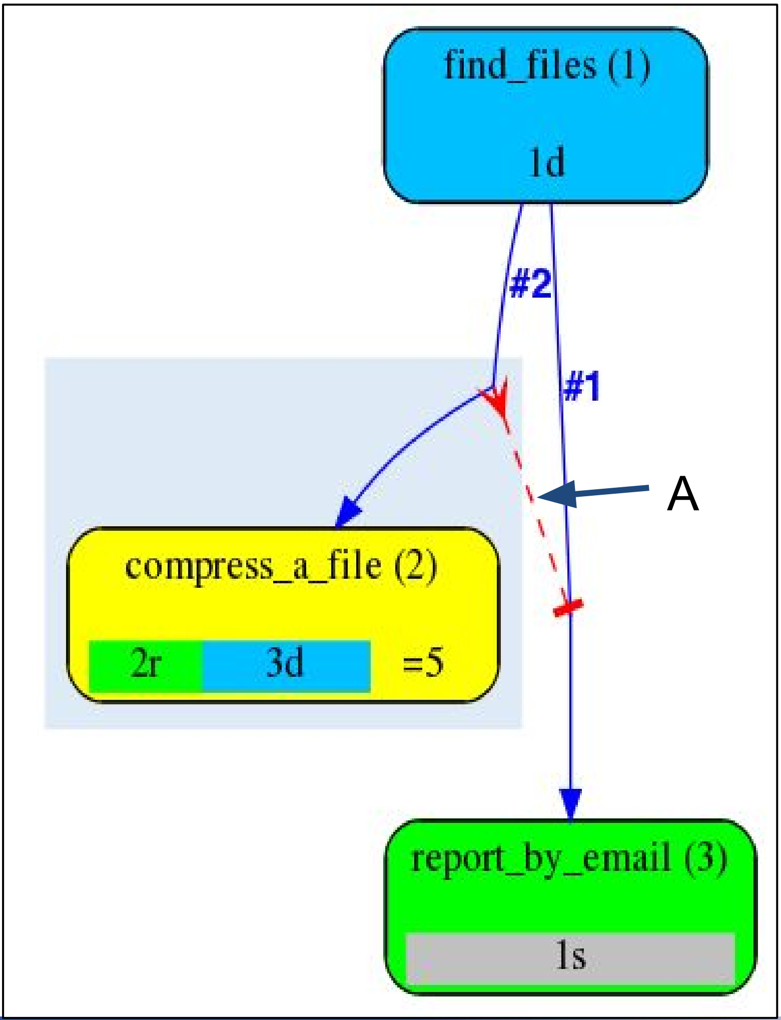
- Built-in mechanism for converging individual threads back together.
- Based on semaphores that can block an individual job by a set of prerequisite jobs.
- Nomenclature: fan (of jobs), funnel (after jobs are done), box (group of jobs in a fan)





Regaining single thread of control: semaphored dataflow

- Syntax for creating a fan and funnel - written into the emitting analysis



- Syntax for creating a fan and funnel: written in the emitting Analysis:

```
-flow_into => {
    '2->A' => [ 'compress_a_file' ],
    'A->1' => [ 'notify' ],
}
```

- In English:
    - For events on branch 2, create a 'compress_a_file' job, and increment the counter on semaphore 'A'
    - For the event on branch 1, create a 'notify' job, which is blocked until the semaphore 'A' count is zero.

Semaphored dataflow in a PipeConfig

- Creating a funnel Analysis : let the pipeline send us a notification:

```
{         -logic_name    => 'notify',
          -module        => 'Bio::EnsEMBL::Hive::RunnableDB::SystemCmd',
          -meadow_type => 'LOCAL',
          -parameters    => {
             'text' => "nothing to report",
             'cmd' => 'notify-send "#text#"',
          },
},
```

- (Reminder) syntax for creating a fan and funnel: written in the emitting Analysis:

```
-flow_into => {
    '2->A' => [ 'compress_a_file' ],
    'A->1' => [ 'notify' ],
}
```

- Exercise - add notification to the compress files pipeline
  - Create a copy CompressFiles_conf.pm called CompressFiles2_conf.pm
    - Be sure to change the module name
  - Add the notify job and wire it in so that it runs after all compress_a_file jobs are finished

Parameters and their implicit propagation

- Exercise 1: How do we set a default directory name for find_files?

- Exercise 2: Adding optional parameters
  - Lets modify our compress files pipeline so that it will only compress files whose names match a particular pattern.
    e.g. if you only want to compress files whose names end with '.fa' you could pass it a wildcard '*.fa' in a parameter. Let's call that parameter 'only_files'
    - Create a copy of CompressFiles2_conf, called CompressFiles3_conf
    - Introduce another parameter **'only_files'** to define the wildcard pattern for filenames we want to compress

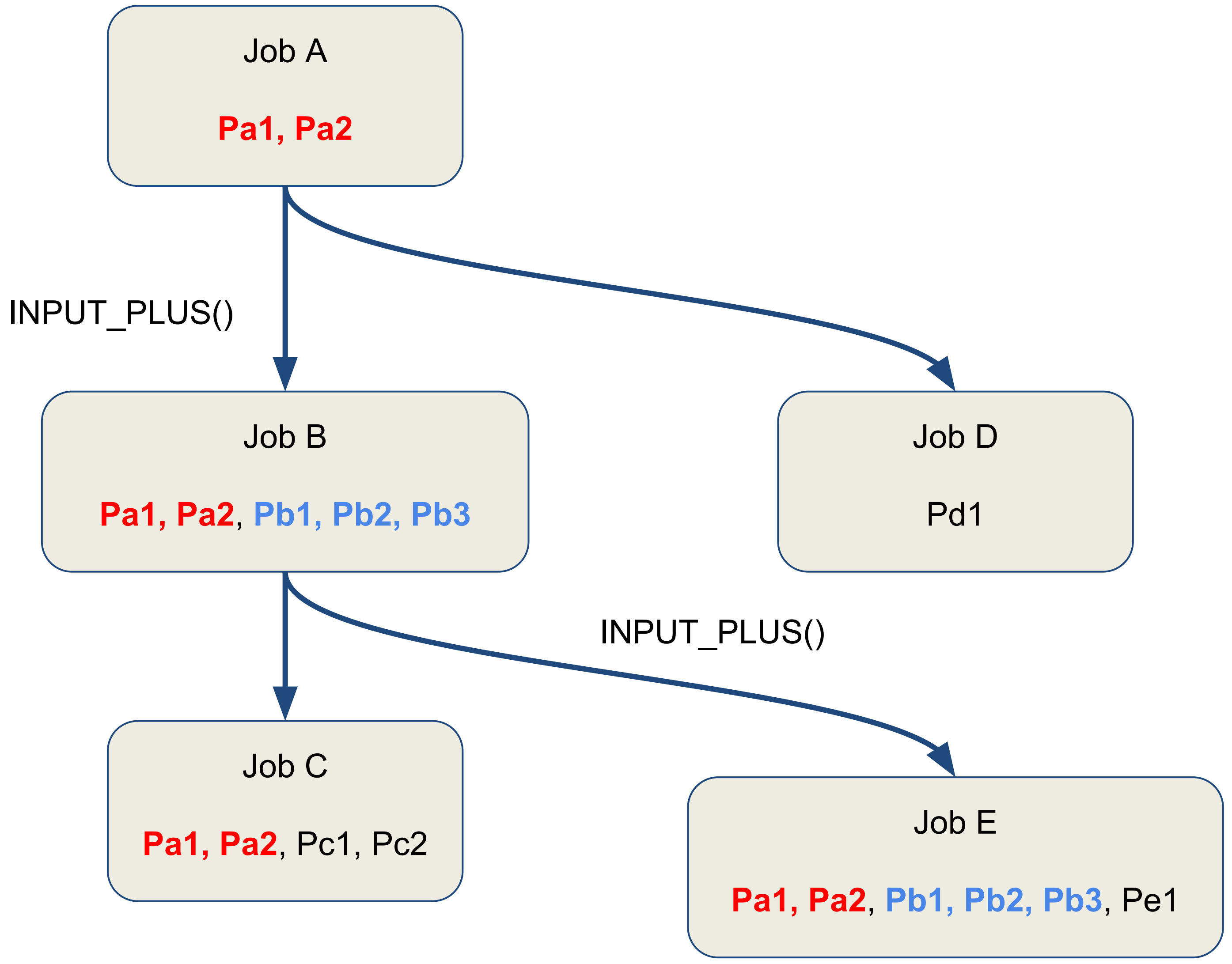
Parameters and their propagation

- What if we wanted to pass something to our compress_a_file analysis directly? Let's say we want the second analysis to work as a compressor or decompressor by putting the appropriate gzip flag into the "cmd" parameter:

```
'cmd'          => 'gzip #gzip_flags# #filename#'
```

- Where do we set gzip_flags?
  - We don't seed any compress_a_file jobs ourselves
- We have to set it when we seed the find_files job.
- That find_files job doesn't know or care about it, but there are ways to pass it along to its children
  - eHive has three main ways of doing this:
    - Explicit (the parameter is written into the emitting analysis' -flow_into)
      ```
      2 => { 'compress_a_file' =>
                {'gzip_flags' => '#gzip_flags#'}}
      ```
    - Implicit (all parameters are available to all children once seen, basically "global parameters")
    - INPUT_PLUS (makes parameters available to all children of a particular analysis selectively)

Using INPUT_PLUS for explicit parameter propagation



Using INPUT_PLUS for explicit parameter propagation

- What if you want automatic parameter propagation in some places, but don't want to turn it on everywhere?
- INPUT_PLUS is a mechanism for propagating all parameters from a parent job to its children, without having to explicitly specify them

```
sub pipeline_analyses {
    my ($self) = @_;
    return [
        {   -logic_name => 'find_files',
            -module     => 'Bio::EnsEMBL::Hive::RunnableDB::JobFactory',
            -parameters => {
                'inputcmd'     => 'find #directory# -type f',
                'column_names' => [ 'filename' ],
            },
            -flow_into => {
                2->A => { 'compress_a_file' => INPUT_PLUS() } ,
                A->1 => [ 'notify' ],
            },
        },

        {   -logic_name    => 'compress_a_file',
            …….. .
        }
```

Note: curly braces {hash} replace brackets [array]

Using INPUT_PLUS for explicit parameter propagation

- Exercise: add a gzip_flags parameter to compress_a_file, and set up propagation from find_files using INPUT_PLUS
  - Create a copy of CompressFiles3_conf.pm, call it CompressFiles4_conf.pm
  - Modify compress_a_file to accept the new parameter in the gzip command
  - Remember the INPUT_PLUS syntax

```
# (in addition to use base)
use Bio::EnsEMBL::Hive::PipeConfig::HiveGeneric_conf;

-flow_into => {
    2->A => { 'compress_a_file' => INPUT_PLUS() } ,
    A->1 => [ 'notify' ],
},
```

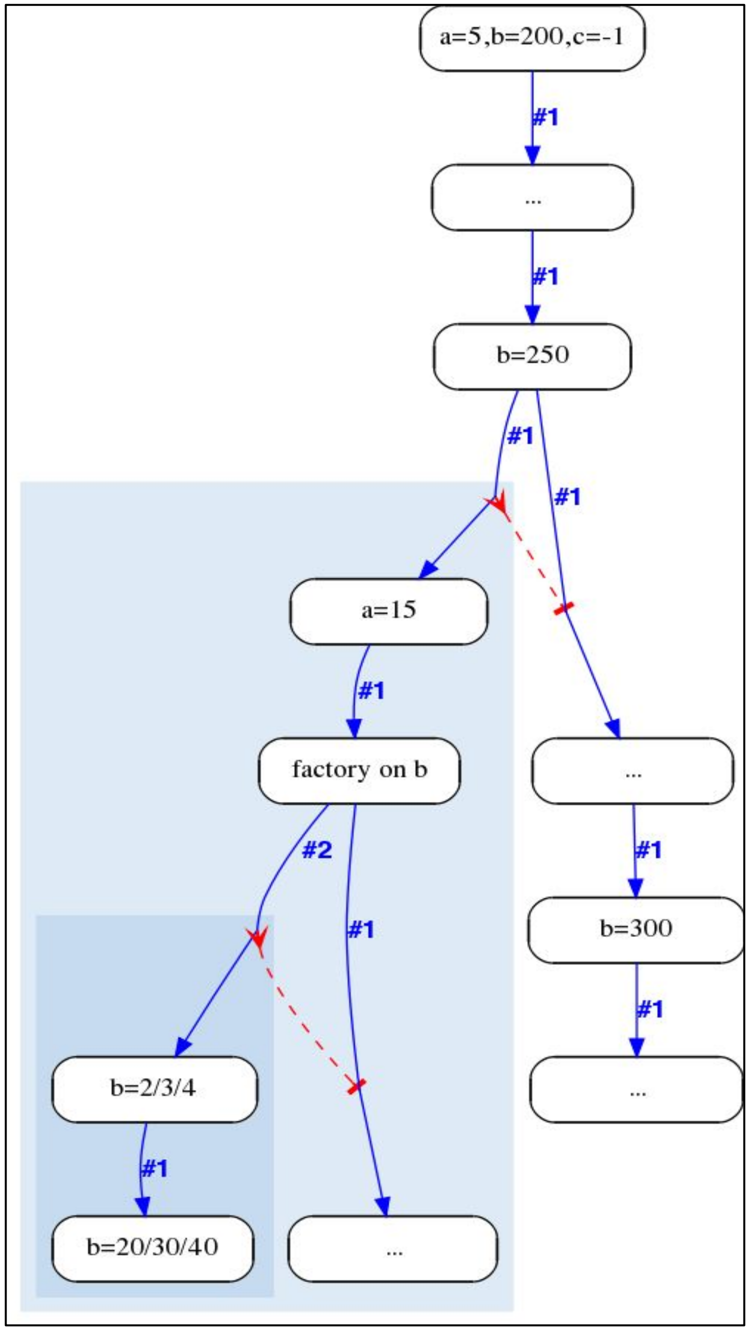
Using implicit parameter propagation

- Implicit parameter propagation is not recommended
  - Was in early versions of hive, and is kept around for backwards compatibility.
  - INPUT_PLUS is recommended.
  - However, you may see it in older pipelines, so this is how it works...
- Implicit propagation is off by default. It is switched on in hive_meta_table like so:

```
sub hive_meta_table {
    my ($self) = @_;
    return {
        %{$self->SUPER::hive_meta_table},
        'hive_use_param_stack'  => 1,
    };
}
```

How implicit parameter propagation works

```
{
my $a=5; my $b=200; my $c=-1;
…
$b=250;
…
          {
               my $a=15;
               for my $b (2,3,4) {
                    $b *= 10;
               }
               …
          }
…
$b=300;
}
```



Capturing data : another role of JobFactory

- SystemCmd (and the similar SqlCmd) only run your command, the output is not captured in any structured way.
- I.e. SqlCmd is usually used to INSERT, DELETE, UPDATE, CREATE/ALTER/DROP TABLE, but not with SELECT.
- JobFactory does capture the output of a command. For a system command, it transforms each line of the output into a dataflow event, and assigns the value in each column to a parameter.
    - (Also can run things like SQL SELECT and convert each line returned into a dataflow event. This Runnable has lots of features that we will only touch on in this course; be sure to check it out when you start writing your own pipelines)
- JobFactory is not specifically creating jobs - it simply transforms streams of “things” into dataflow events that may be converted into Jobs, stored in database tables, or accumulated. It is the wiring that defines what happens next.
    - Also, it isn't specifically creating fans of jobs with a funnel. The fan-funnel relationship is determined by how subsequent jobs are wired in -flow_into

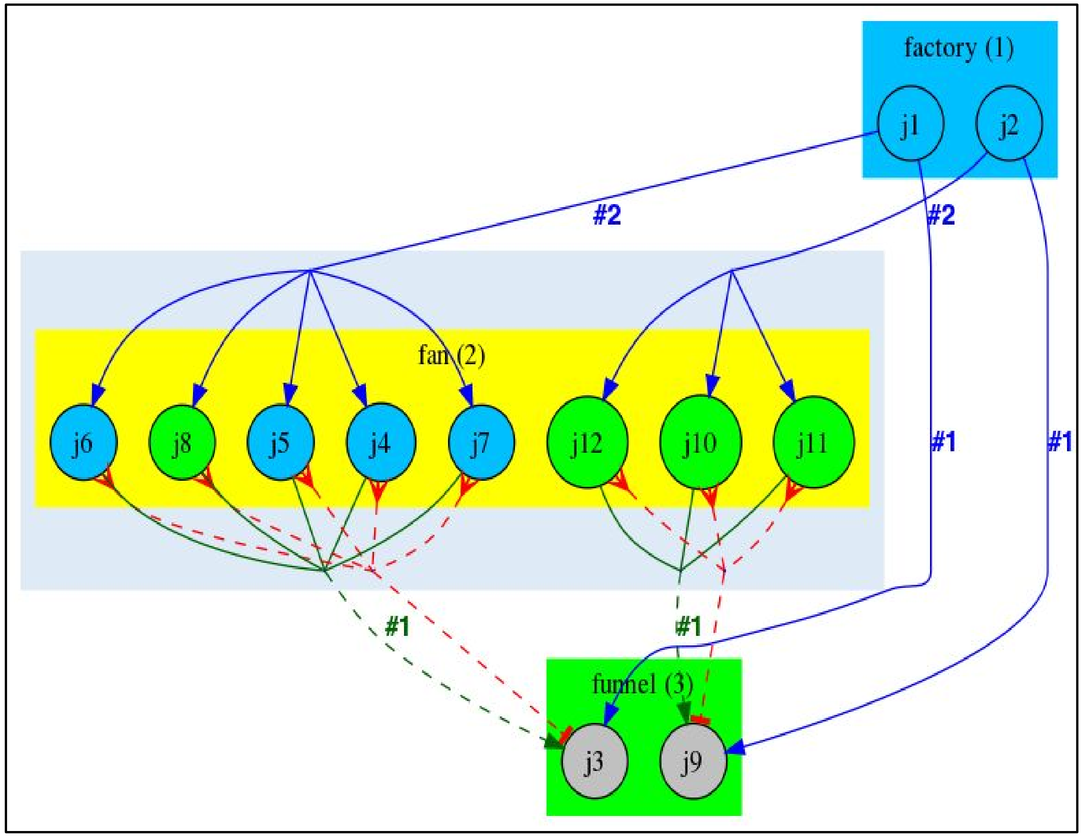
Capturing data : another role of JobFactory

- Here is an example of using JobFactory to capture the output from a command that produces two columns.
  - Try running `wc -c` on a file to see what the output looks like
    - You can trim off the leading whitespace by piping the output into `sed -e 's/^ *//'`
  - Notice there are two columns, so if we run this command in JobFactory, there will be two parameters to capture -- one for each column.
    - This is why we set the column_names parameter to an array - we can provide a parameter name for each column of output that the inputcmd command generates
  - Also, because you're specifying a specific filename, the command produces only one line of output. So when JobFactory runs it, it will produce only one event.

```
{    -logic_name    => 'pre_compress_size',
     -module        => 'Bio::EnsEMBL::Hive::RunnableDB::JobFactory',
     -parameters    => {
         'inputcmd'       => "wc -c #filename# | sed -e 's/^ *//' ",
         'delimiter'      => ' ',
         'column_names'   => [ 'orig_size', 'orig_filename' ],
     },
},
```

Accumulating data from a semaphore group

- How do we pass the data from the box into the funnel?
- The data can be passed from any job within the box into the correct funnel Job
- Different structures or combinations can be accumulated (hashes, arrays, piles, multisets)
- Accumulator URLs as targets for Dataflow.



Accumulating data from a semaphore group

- Accumulator syntax:

```
'?accu_name=hash_name&accu_address={key_name}'

'?accu_name=array_name&accu_address=[index_name]'

'?accu_name=pile_name&accu_address=[]'

'?accu_name=multiset_name&accu_address={}'

'?accu_name=pile_name&accu_address=[]&accu_input_variable=pile_component'
```
(dataflow pile_component and they get assembled into pile called pile_name)

```
'?accu_name=scalar_name'
```

example:
Let's say you are flowing out paramaters called "size" and "filename". If you flow them into this accu:

```
'?accu_name=size&accu_address={filename}'
```

You will end up with a hash named "size" where the key is the value of "filename" and the value is the value of "size"

- see GCPct_conf and LongMult_conf for examples of a PipeConfig file with an accu

Analogous Perl expressions for accus

- Hash accu (like a Perl hash):

```
?accu_name=filename&accu_address={species}
```

In Perl

```
my $filename = "foo.fa";
my $species = "Mus musculus";
my %filename;
$filename{$species} = $filename;
```

using accu_input_variable

```
?accu_name=paths&accu_address={species}&accu_input_variable=filename
```

In Perl

```
my $filename = "foo.fa";
my $species = "Mus musculus";
my %paths;
$paths{$species} = $filename;
```

Analogous Perl expressions for accus

- Array accu (like Perl array):

```
?accu_name=filename&accu_address=[chunk]
```

In Perl

```
my $filename = "foo.fa";
my $chunk = 2;
my @filename;
$filename[$chunk] = $filename;
```

using accu_input_variable

```
?accu_name=paths&accu_address=[chunk]&accu_input_variable=filename
```

In Perl

```
my $filename = "foo.fa";
my $chunk = 2;
my @paths;
$paths[$chunk] = $filename;
```

Analogous Perl expressions for accus

- Pile accu (no direct equivalent in Perl):

```
?accu_name=filename&accu_address=[]
```

In Perl

```
my $filename = "foo.fa";
push(@filename, $filename);
```

using accu_input_variable

```
?accu_name=paths&accu_address=[]&accu_input_variable=filename
```

In Perl

```
my $filename = "foo.fa";
push (@paths, $filename);
```

Analogous Perl expressions for accus

- Multiset accu (hash as a counter trick in Perl):

```
?accu_name=transcripts&accu_address={}
```

In Perl

```
my @list_of_transcripts = (.....);
my %transcripts = {};
foreach my $transcript (@list_of_transcripts) {
    $transcripts{$transcript} += 1;
}
```

using accu_input_variable

```
?accu_name=xcript_count&accu_address={}&accu_input_variable=transcript
```

In Perl

```
foreach my $transcript (@list_of_transcripts) {
    $xcript_count{$transcript} += 1;
}
```

Advanced parameter substitution : expressions

- You can manipulate the values of parameters using any Perl expression by wrapping the expression within #expr( )expr#
- For example, to add 1 to the value in parameter 'alpha'

```
'alpha_plus_one' => '#expr( #alpha#+1 )expr#'
```

- What's going on above:
  - First, #alpha# gets text-substituted with the value of alpha
  - Then, the resulting string is eval-ed in Perl.
- For standard Perl interpretation of a variable, add dollarsign + space before the variable name

```
$ beta
```

- If the parameter holds a data structure (arrayref or hashref), you can dereference it with curly-braces like in standard Perl

```
@{ #array_ref# }
%{ #hash_ref }
```

Advanced parameter substitution : expressions

- We can reduce accumulated structures (that are not scalars) into scalars using #expr()expr# . For example,

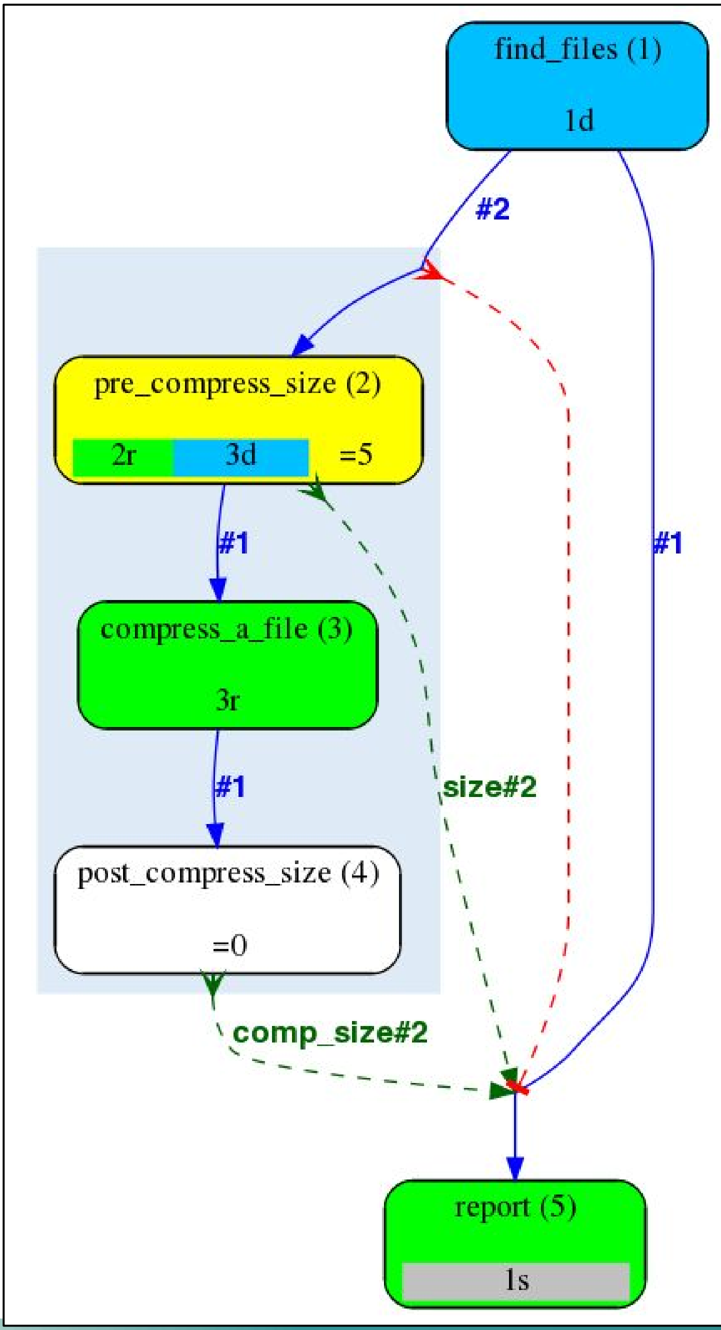
```
'min_comp_size' => '#expr(min values %{#comp_size#})expr#',

'max_comp_size' => '#expr(max values %{#comp_size#})expr#',

'text'     => 'compressed sizes between #min_comp_size# and #max_comp_size#',
```

Exercise: accumulation + substitution

- Let's put it all together:
  - Factory on a directory to dataflow single filenames
  - compute their sizes and accumulate them
  - compress the files
  - compute the compressed sizes and accumulate
  - funnel reduces the accumulated structures and sends a notification
- Solution: CompressFiles5_conf



Conditional dataflow

- It is possible to direct events based on values of parameters
- WHEN - ELSE syntax
  - Similar to IF - THEN, but with important differences
  - WHEN happens when the condition is true
  - Can be multiple WHEN cases, and more than one WHEN can flow as long as they are true
  - ELSE is the catch-all if none of the WHEN cases are true

```
-flow_into => { 1 => [ WHEN('#a# > 3'  => ['analysis_b'],
                            '#a# > 5'  => ['analysis_c'],
                               ELSE     ['analysis_d']),
                        'analysis_e'
                      ],
              }
```

value of a...	events creating jobs for...
2	d, e
4	b, e
6	b, c, e

Questions?

Ensembl Acknowledgements

The Entire Ensembl Team

Bronwen L. Aken[1], Premanand Achuthan[1], Wasiu Akanni[1], M. Ridwan Amode[1], Friederike Bernsdorff[1], Jyothish Bhai[1], Konstantinos Billis[1], Denise Carvalho-Silva[1], Carla Cummins[1], Peter Clapham[2], Laurent Gil[1], Carlos García Girón[1], Leo Gordon[1], Thibaut Hourlier[1], Sarah E. Hunt[1], Sophie H. Janacek[1], Thomas Juettemann[1], Stephen Keenan[1], Matthew R. Laird[1], Ilias Lavidas[1], Thomas Maurel[1], William McLaren[1], Benjamin Moore[1], Daniel N. Murphy[1], Rishi Nag[1], Victoria Newman[1], Michael Nuhn[1], Chuang Kee Ong[1], Anne Parker[1], Mateus Patricio[1], Harpreet Singh Riat[1], Daniel Sheppard[1], Helen Sparrow[1], Kieron Taylor[1], Anja Thormann[1], Alessandro Vullo[1], Brandon Walts[1], Steven P. Wilder[1], Amonida Zadissa[1], Myrto Kostadima[1], Fergal J. Martin[1], Matthieu Muffato[1], Emily Perry[1], Magali Ruffier[1], Daniel M. Staines[1], Stephen J. Trevanion[1], Fiona Cunningham[1], Andrew Yates[1], Daniel R. Zerbino[1] and Paul Flicek[1,2,*]

[1]European Molecular Biology Laboratory, European Bioinformatics Institute, Wellcome Genome Campus, Hinxton, Cambridge CB10 1SD, UK and [2]Wellcome Trust Sanger Institute, Wellcome Genome Campus, Hinxton, Cambridge, CB10 1SA, UK

Funding

wellcome

EMBL

National Human Genome Research Institute

Open Targets

BBSRC bioscience for the future

TRANSFORMING GENETIC MEDICINE INITIATIVE

MedBioinformatics

Co-funded by the European Union