

AN INTRODUCTION TO WORKFLOWS WITH **DAGMAN**

Presented by Jaime Frey

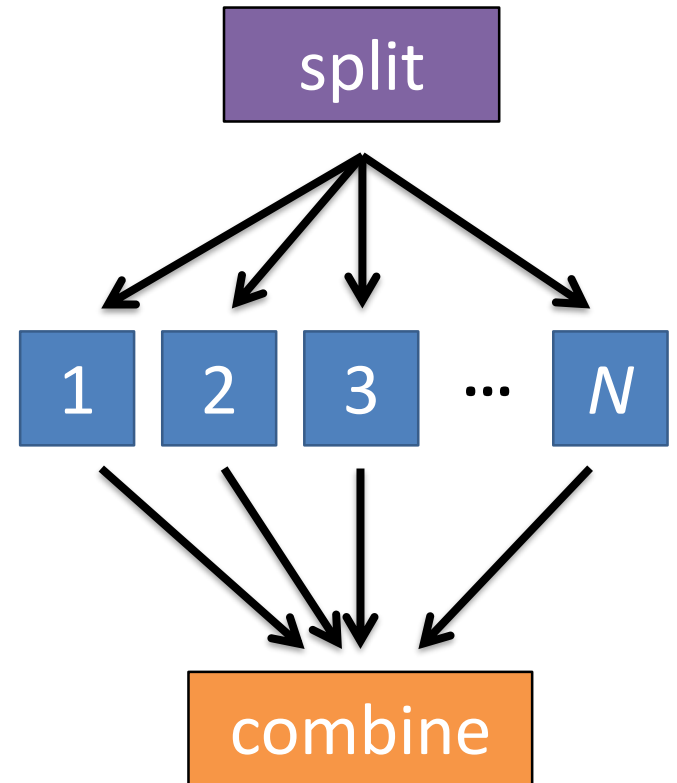
Covered In This Tutorial

- Why Create a Workflow?
- Describing workflows as *directed acyclic graphs* (DAGs)
- Workflow execution via DAGMan (DAG Manager)
- Node-level options in a DAG
- Modular organization of DAG components
- DAG-level control
- Additional DAGMan Features

Why Workflows? Why “DAGs”?

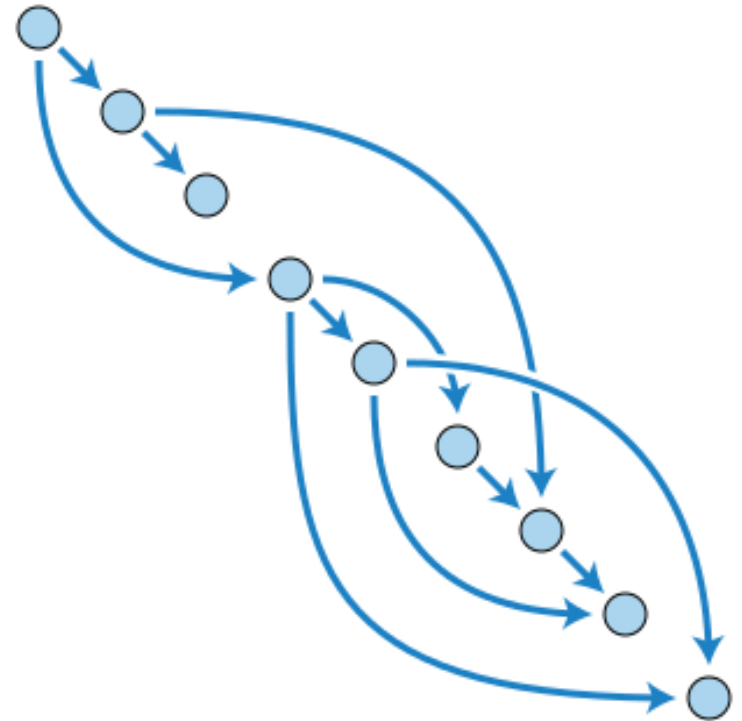
Automation!

- Objective: Submit jobs in a particular order, *automatically*.
- Especially if: Need to reproduce the same workflow multiple times.



DAG = "directed acyclic graph"

- topological ordering of vertices ("nodes") is established by directional connections ("edges")
- "acyclic" aspect requires a start and end, with no looped repetition
 - can contain cyclic subcomponents, covered in later slides for workflows

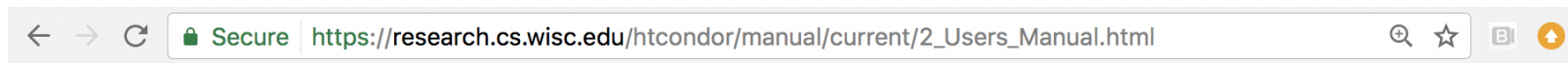


Wikimedia Commons

wikipedia.org/wiki/Directed_acyclic_graph

Describing Workflows with DAGMan

DAGMan in the HTCondor Manual

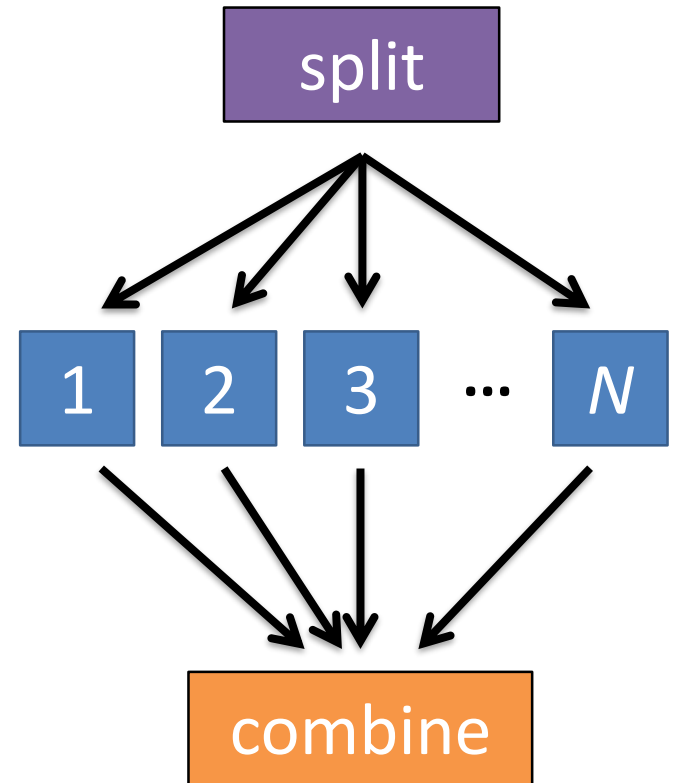


- [2.9.2 Parallel Jobs and the Dedicated Scheduler](#)
- [2.9.3 Submission Examples](#)
- [2.9.4 MPI Applications Within HTCondor's Vanilla Universe](#)

- [2.10 DAGMan Applications](#)
 - [2.10.1 DAGMan Terminology](#)
 - [2.10.2 The DAG Input File: Basic Commands](#)
 - [2.10.3 Command Order](#)
 - [2.10.4 Node Job Submit File Contents](#)
 - [2.10.5 DAG Submission](#)
 - [2.10.6 File Paths in DAGs](#)
 - [2.10.7 DAG Monitoring and DAG Removal](#)
 - [2.10.8 Suspending a Running DAG](#)
 - [2.10.9 Advanced Features of DAGMan](#)
 - [2.10.10 The Rescue DAG](#)
 - [2.10.11 DAG Recovery](#)
 - [2.10.12 Visualizing DAGs with *dot*](#)
 - [2.10.13 Capturing the Status of Nodes in a File](#)
 - [2.10.14 A Machine-Readable Event History, the *jobstate.log* File](#)
 - [2.10.15 Status Information for the DAG in a ClassAd](#)
 - [2.10.16 Utilizing the Power of DAGMan for Large Numbers of Jobs](#)
 - [2.10.17 Workflow Metrics](#)
 - [2.10.18 DAGMan and Accounting Groups](#)

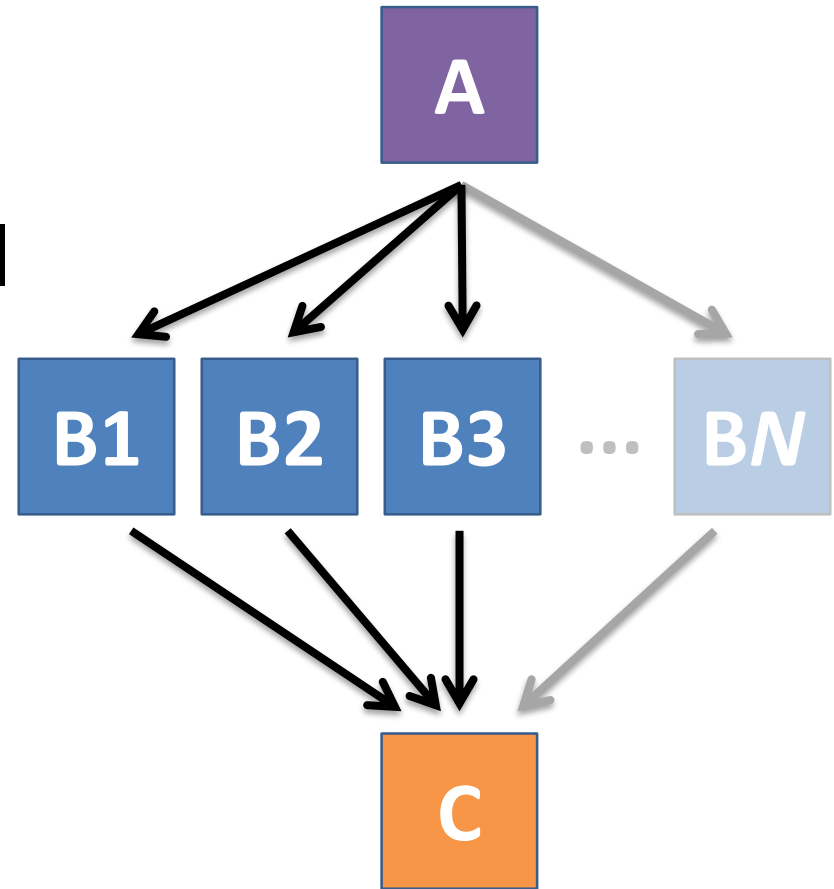
An Example HTC Workflow

- User must communicate the “**nodes**” and directional “**edges**” of the DAG



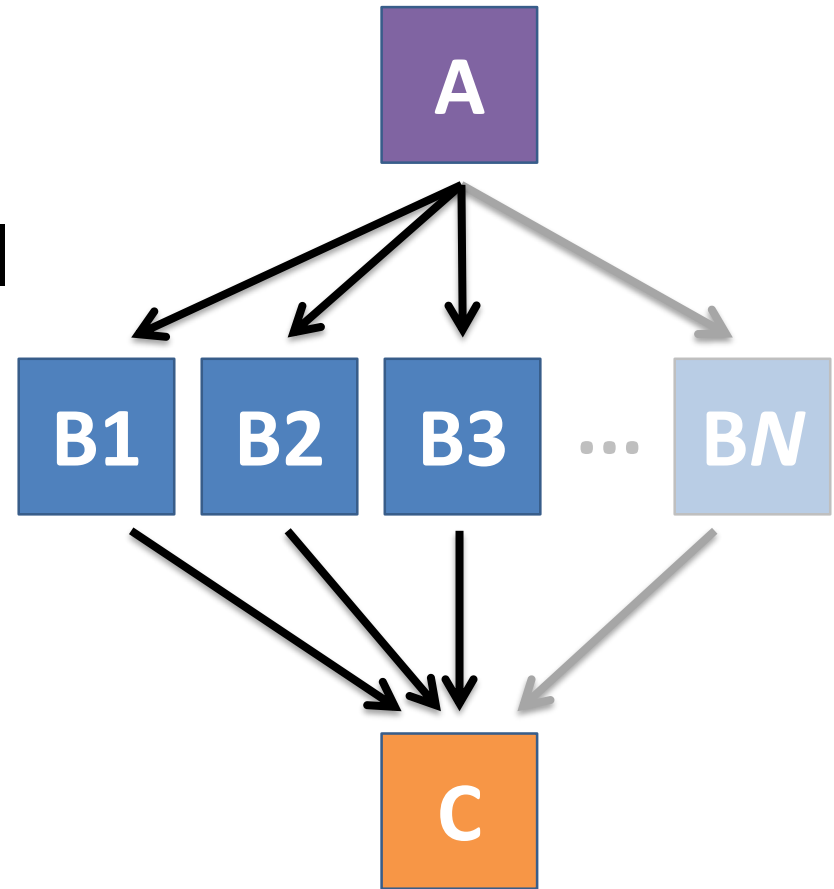
Simple Example for this Tutorial

- **The DAG input file** communicates the “nodes” and directional “edges” of the DAG



Simple Example for this Tutorial

- **The DAG input file** communicates the “nodes” and directional “edges” of the DAG



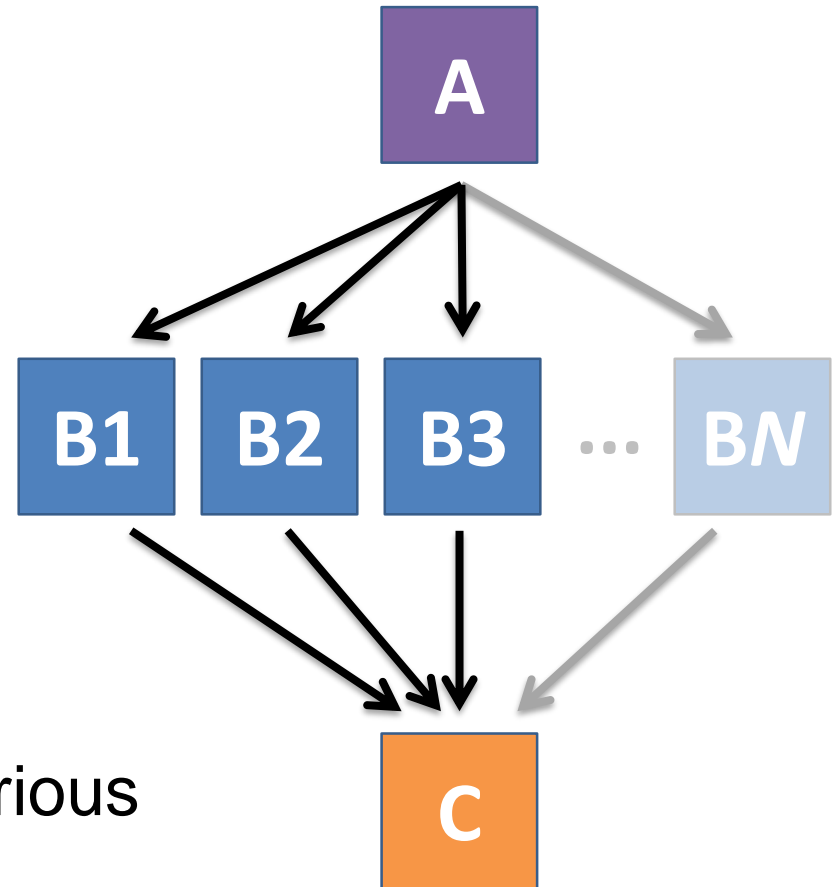
Look for links on future slides



Basic DAG input file: JOB nodes, PARENT-CHILD edges

my.dag

```
JOB A A.sub
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C
```



- Node names are used by various DAG features to modify their execution by DAG Manager.

Basic DAG input file: JOB nodes, PARENT-CHILD edges

my.dag

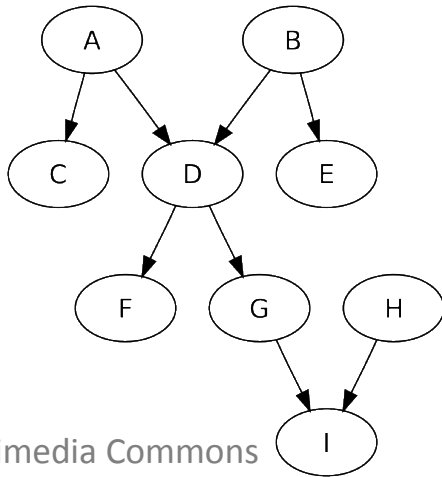
```
JOB A A.sub
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C
```

(dag_dir) /

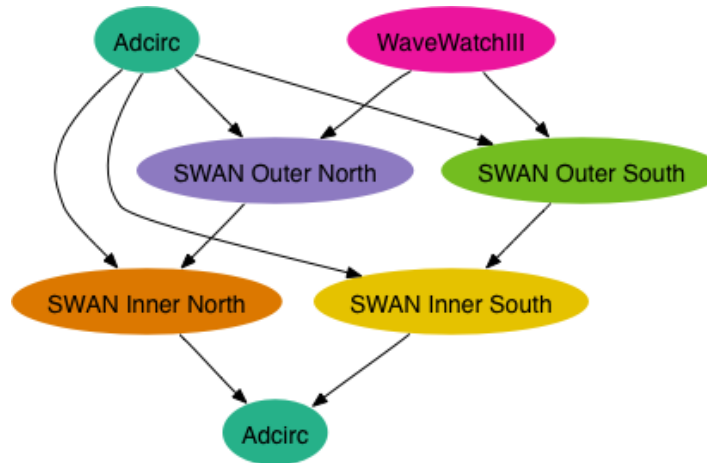
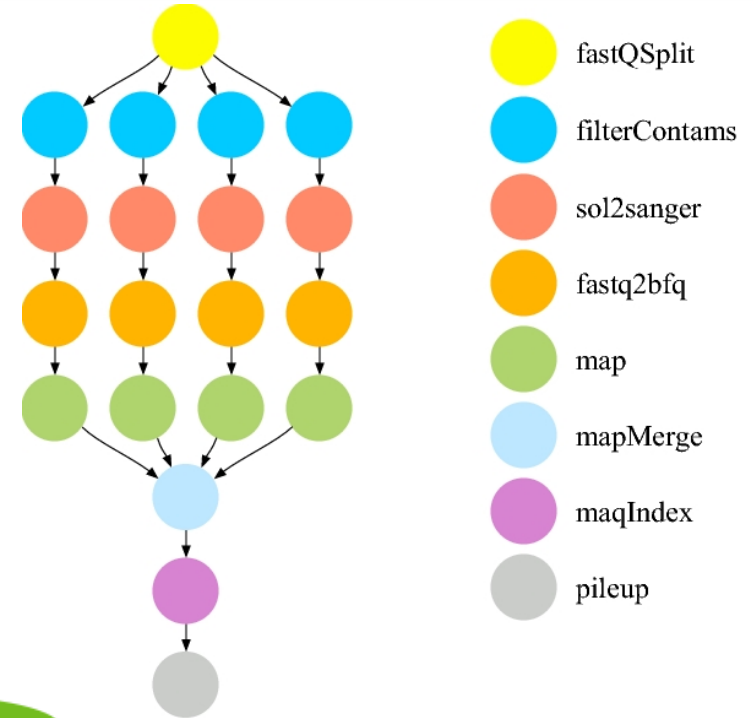
```
A.sub      B1.sub
B2.sub     B3.sub
C.sub      my.dag
(other job files)
```

- Node names and filenames can be anything.
- Node name and submit filename do not have to match.

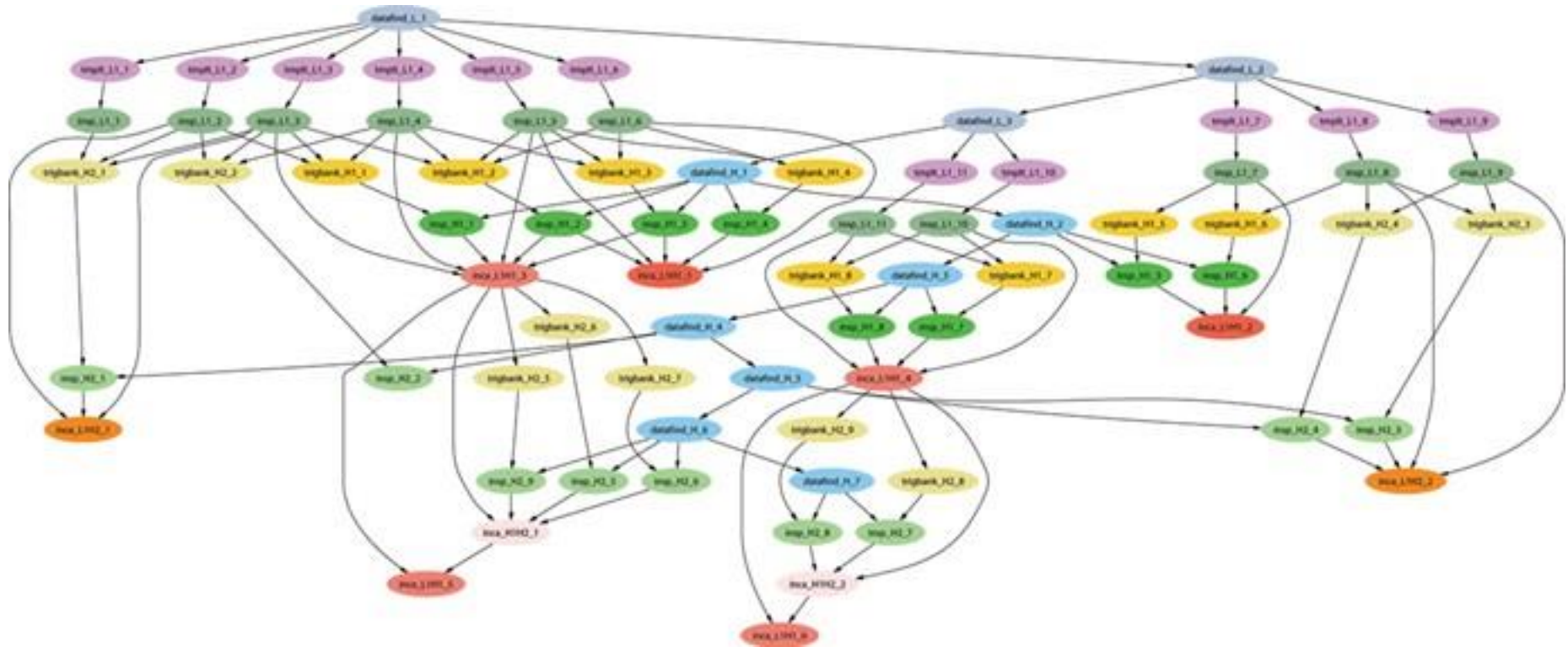
Endless Workflow Possibilities



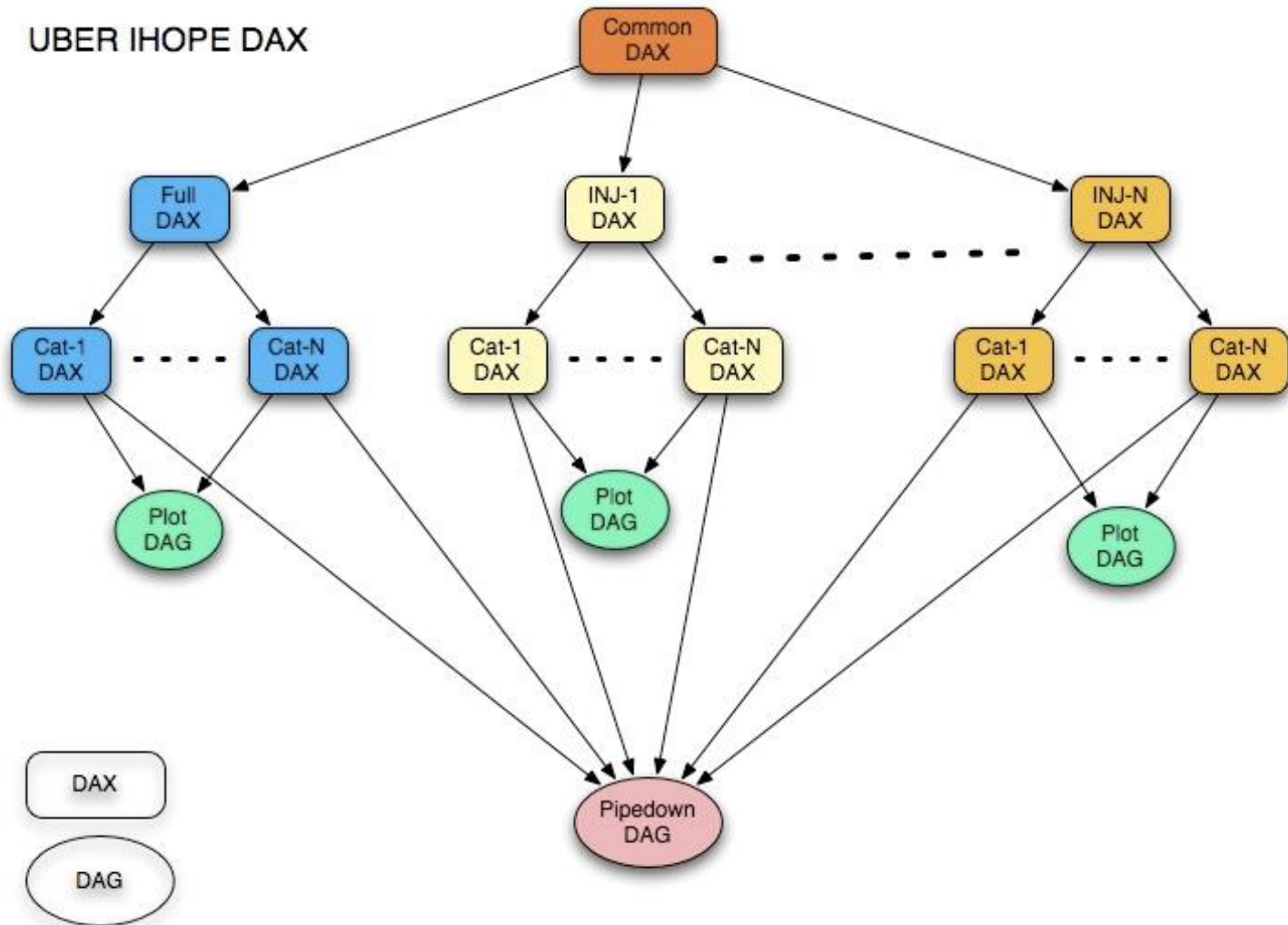
Wikimedia Commons



Endless Workflow Possibilities

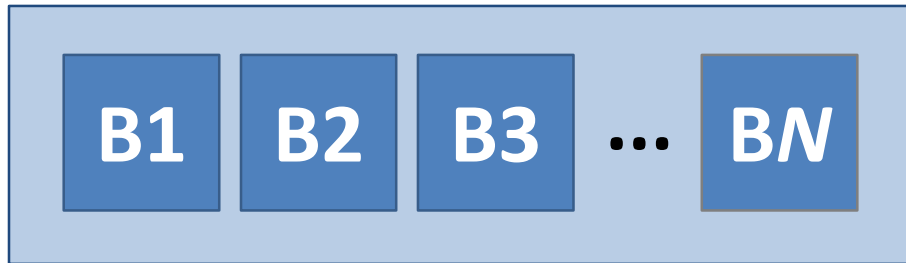


Repeating DAG Components!!

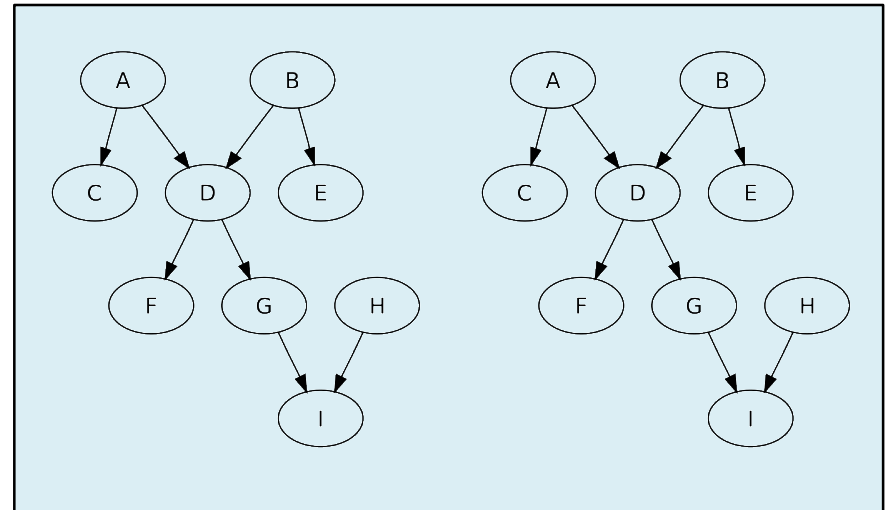


DAGs are also useful for non-sequential work

'bag' of HTC jobs



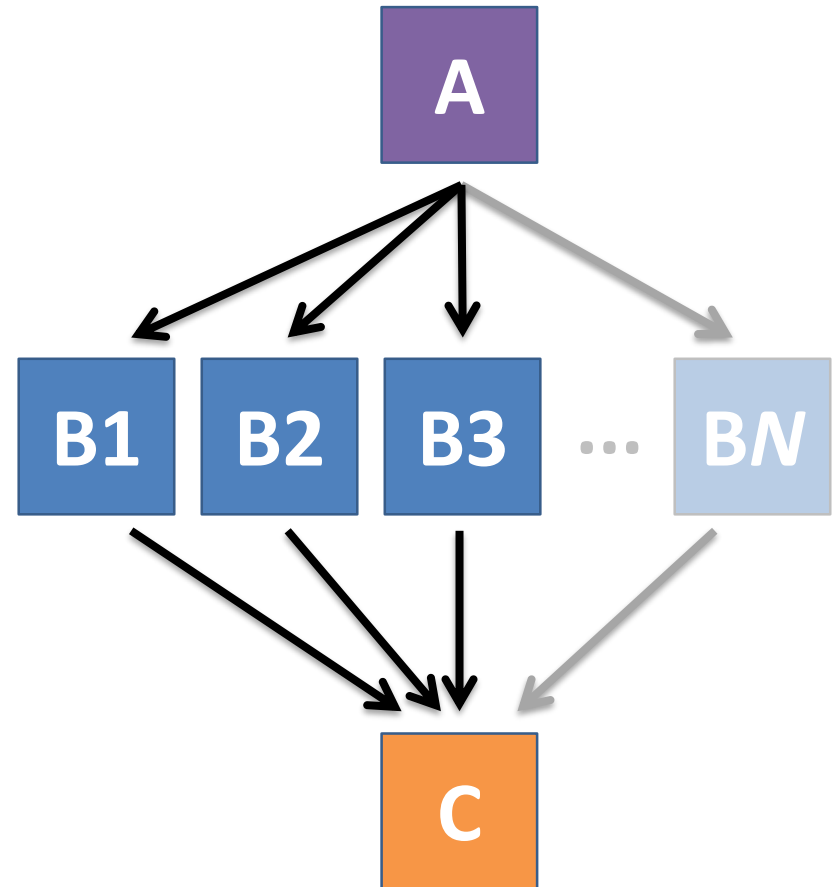
disjoint workflows



Basic DAG input file: JOB nodes, PARENT-CHILD edges

my.dag

```
JOB A A.sub  
JOB B1 B1.sub  
JOB B2 B2.sub  
JOB B3 B3.sub  
JOB C C.sub  
PARENT A CHILD B1 B2 B3  
PARENT B1 B2 B3 CHILD C
```



Submitting and Monitoring a DAGMan Workflow

Submitting a DAG to the queue

- Submission command:

`condor_submit_dag dag_file`

```
$ condor_submit_dag my.dag
```

```
-----  
File for submitting this DAG to HTCondor           : my.dag.condor.sub  
Log of DAGMan debugging messages                  : my.dag.dagman.out  
Log of HTCondor library output                    : my.dag.lib.out  
Log of HTCondor library error messages            : my.dag.lib.err  
Log of the life of condor_dagman itself           : my.dag.dagman.log
```

```
Submitting job(s).
```

```
1 job(s) submitted to cluster 87274940.
```

```
-----
```

A submitted DAG creates a DAGMan job process in the queue

- DAGMan runs on the submit server, as a job in the queue
- **At first:**

```
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
OWNER      BATCH_NAME      SUBMITTED      DONE      RUN      IDLE      TOTAL      JOB_IDS
alice      my.dag+128      4/30 18:08
1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended

$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
ID         OWNER      SUBMITTED      RUN_TIME  ST PRI  SIZE  CMD
128.0     alice     4/30 18:08     0+00:00:06 R  0    0.3  condor_dagman
1 jobs; 0 completed, 0 removed, 0 idle, 1 running, 0 held, 0 suspended
```

Jobs are automatically submitted by the DAGMan job

- Seconds later, node **A** is submitted:

```
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
OWNER    BATCH_NAME    SUBMITTED    DONE    RUN    IDLE    TOTAL    JOB_IDS
alice    my.dag+128    4/30 18:08    _      1      5    129.0
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended

$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
ID       OWNER      SUBMITTED      RUN_TIME    ST PRI  SIZE  CMD
128.0    alice     4/30 18:08     0+00:00:36 R  0     0.3  condor_dagman
129.0    alice     4/30 18:08     0+00:00:00 I  0     0.3  A_split.sh
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended
```

Jobs are automatically submitted by the DAGMan job

- After **A** completes, **B1-3** are submitted

```
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
OWNER    BATCH_NAME    SUBMITTED    DONE    RUN    IDLE    TOTAL    JOB_IDS
alice    my.dag+128    4/30 18:08    1      3      5    130.0 ... 132.0
4 jobs; 0 completed, 0 removed, 3 idle, 1 running, 0 held, 0 suspended
```

```
$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
  ID      OWNER      SUBMITTED      RUN_TIME  ST  PRI  SIZE  CMD
128.0    alice     4/30 18:08    0+00:20:36  R   0    0.3  condor_dagman
130.0    alice     4/30 18:28    0+00:00:00  I   0    0.3  B_run.sh
131.0    alice     4/30 18:28    0+00:00:00  I   0    0.3  B_run.sh
132.0    alice     4/30 18:28    0+00:00:00  I   0    0.3  B_run.sh
4 jobs; 0 completed, 0 removed, 3 idle, 1 running, 0 held, 0 suspended
```

Jobs are automatically submitted by the DAGMan job

- After **B1-3** complete, node **C** is submitted

```
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
OWNER    BATCH_NAME    SUBMITTED    DONE    RUN    IDLE    TOTAL    JOB_IDS
alice    my.dag+128    4/30 18:08     4      1      5    133.0
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended

$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
ID       OWNER    SUBMITTED    RUN_TIME  ST PRI  SIZE  CMD
128.0    alice    4/30 18:08    0+00:46:36 R  0     0.3  condor_dagman
133.0    alice    4/30 18:54    0+00:00:00 I  0     0.3  C_combine.sh
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended
```

Status files are Created at the time of DAG submission

(dag_dir) /

A.sub	B1.sub	B2.sub
B3.sub	C.sub	<i>(other job files)</i>
my.dag	my.dag.condor.sub	my.dag.dagman.log
my.dag.dagman.out	my.dag.lib.err	my.dag.lib.out
my.dag.nodes.log		

- * **.condor.sub** and * **.dagman.log** describe the queued DAGMan job process, as for all queued jobs
- * **.dagman.out** has detailed logging (look to first for errors)
- * **.lib.err/out** contain std err/out for the DAGMan job process
- * **.nodes.log** is a combined log of all jobs within the DAG

Removing a DAG from the queue

- Remove the DAGMan job in order to stop and remove the entire DAG:

condor_rm dagman_jobID

```
$ condor_q
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
OWNER   BATCH_NAME   SUBMITTED   DONE   RUN   IDLE   TOTAL   JOB_IDS
alice   my.dag+128   4/30 18:08     4     _     1     6   133.0
2 jobs; 0 completed, 0 removed, 1 idle, 1 running, 0 held, 0 suspended
$ condor_rm 128
All jobs in cluster 128 have been marked for removal
```

- Creates a **rescue file** so that only incomplete or unsuccessful NODES are repeated upon resubmission

[DAGMan > DAG Monitoring and DAG Removal](#)
[DAGMan > The Rescue DAG](#)

Removal of a DAG results in a rescue file

(dag_dir) /

```
A.sub  B1.sub  B2.sub  B3.sub  C.sub  (other job files)
my.dag                my.dag.condor.sub  my.dag.dagman.log
my.dag.dagman.out    my.dag.lib.err      my.dag.lib.out
my.dag.metrics       my.dag.nodes.log   my.dag.rescue001
```

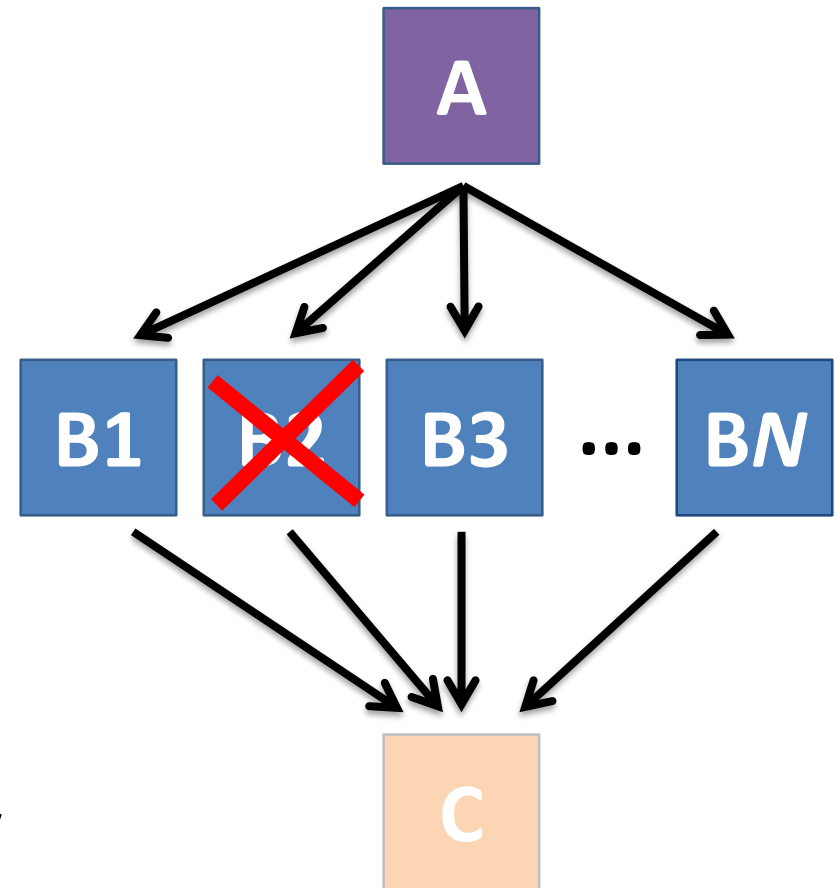
- Named ***dag_file.rescue001***
 - increments if more rescue DAG files are created
- Records which NODES have completed successfully
 - does not contain the actual DAG structure

Rescue Files For Resuming a Failed DAG

- A **rescue file** is created any time a DAG is removed from the queue by the user (`condor_rm`) or automatically:
 - a **node fails**, and after DAGMan advances through any other possible nodes
 - the DAG is **aborted** (covered later)
 - the DAG is **halted** and not unhalted (covered later)
- The **rescue file** will be used (**if it exists**) when the original DAG file is resubmitted
 - override: `condor_submit_dag dag_file -f`

Node Failures Result in DAG Failure and Removal

- **If a node JOB fails** (non-zero exit code)
 - DAGMan continues to run other JOB nodes until it can no longer make progress
- Example at right:
 - **B2** fails
 - Other **B*** jobs continue
 - DAG fails and exits after **B*** and before node **C**



Resolving held node jobs

```
$ condor_q -nobatch
-- Schedd: submit-3.chtc.wisc.edu : <128.104.100.44:9618?...
  ID      OWNER    SUBMITTED    RUN_TIME  ST  PRI  SIZE  CMD
128.0    alice    4/30 18:08    0+00:20:36 R   0     0.3  condor_dagman
130.0    alice    4/30 18:18    0+00:00:00 H   0     0.3  B_run.sh
131.0    alice    4/30 18:18    0+00:00:00 H   0     0.3  B_run.sh
132.0    alice    4/30 18:18    0+00:00:00 H   0     0.3  B_run.sh
4 jobs; 0 completed, 0 removed, 0 idle, 1 running, 3 held, 0 suspended
```

- Look at the hold reason (in the job log, or with `'condor_q -hold'`)
- Fix the issue and release the jobs (`condor_release`)
-OR- remove the entire DAG, resolve, then resubmit the DAG

DAG Completion

(dag_dir) /

A.sub	B1.sub	B2.sub
B3.sub	C.sub	<i>(other job files)</i>
my.dag	my.dag.condor.sub	my.dag.dagman.log
my.dag.dagman.out	my.dag.lib.err	my.dag.lib.out
my.dag.nodes.log	my.dag.dagman.metrics	

- * **.dagman.metrics** is a summary of events and outcomes
- * **.dagman.log** will note the completion of the DAGMan job
- * **.dagman.out** has detailed logging for all jobs (look to first for errors)

Beyond the Basic DAG: Node-level Modifiers

Default File Organization

my.dag

```
JOB A A.sub
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C
```

(dag_dir) /

```
A.sub      B1.sub
B2.sub    B3.sub
C.sub      my.dag
(other job files)
```

- What if you want to organize files in other directories?

Node-specific File Organization with DIR

- **DIR** sets the submission directory of the node

my.dag

```
JOB A A.sub DIR A
JOB B1 B1.sub DIR B
JOB B2 B2.sub DIR B
JOB B3 B3.sub DIR B
JOB C C.sub DIR C
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C
```

(dag_dir) /

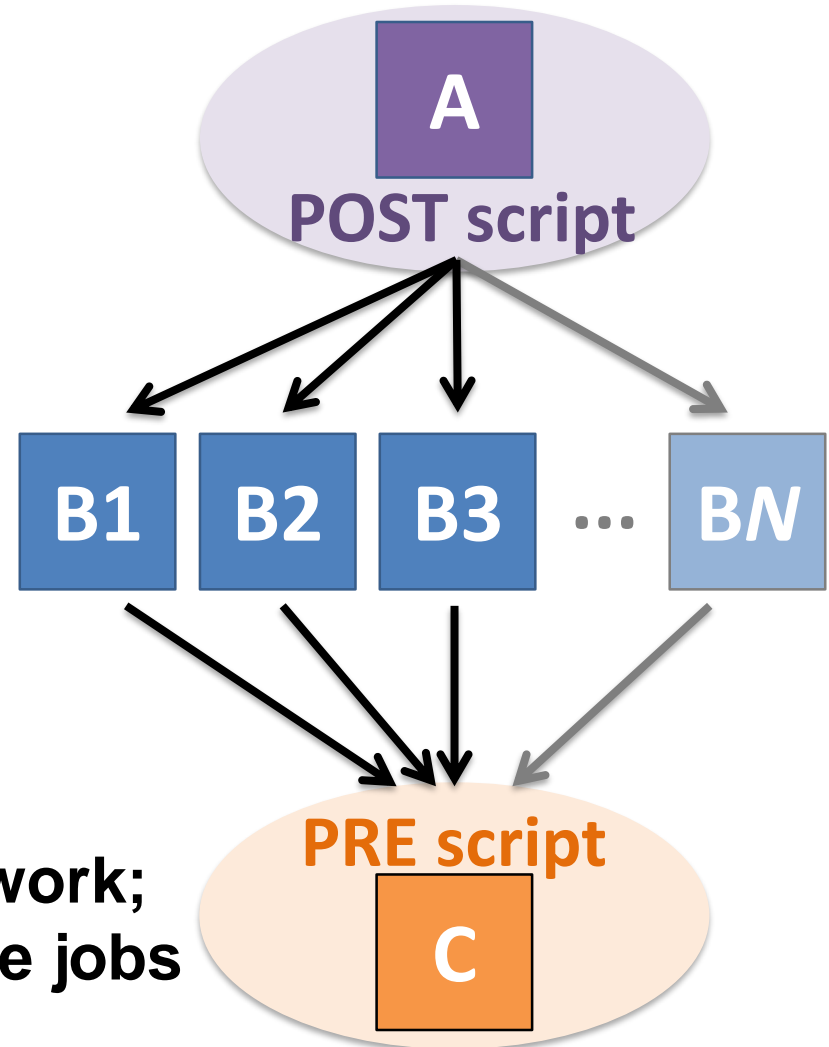
```
my.dag
A/ A.sub (A job files)
B/ B1.sub B2.sub
   B3.sub (B job files)
C/ C.sub (C job files)
```

PRE and POST scripts run on the submit server, as part of the node

my.dag

```
JOB A A.sub
SCRIPT POST A sort.sh
JOB B1 B1.sub
JOB B2 B2.sub
JOB B3 B3.sub
JOB C C.sub
SCRIPT PRE C tar_it.sh
PARENT A CHILD B1 B2 B3
PARENT B1 B2 B3 CHILD C
```

- **Use sparingly for lightweight work; otherwise include work in node jobs**



RETRY failed nodes to overcome transient errors

- Retry a node up to N times if it fails (the job exit code is non-zero):

RETRY *node_name* N

Example:

```
JOB A A.sub
RETRY A 5
JOB B B.sub
PARENT A CHILD B
```

- See also: `retry` except for a particular exit code (`UNLESS-EXIT`)
- **Note:** `max_retries` in the submit file are preferable for simple cases

[DAGMan Applications > Advanced Features > Retrying](#)

[DAGMan Applications > DAG Input File > SCRIPT](#)

RETRY applies to whole node, including PRE/POST scripts

- PRE and POST scripts are included in retries
- **RETRY of a node with a POST script uses the exit code from the POST script (not from the job)**
 - POST script can do more to determine node success, perhaps by examining JOB output

Example:

```
SCRIPT PRE A download.sh
JOB A A.sub
SCRIPT POST A checkA.sh
RETRY A 5
```

SCRIPT Arguments and Argument Variables

```
JOB A A.sub  
SCRIPT POST A checkA.sh my.out $RETURN  
RETRY A 5
```

\$JOB: node name

\$JOBID: *cluster.proc*

\$RETURN: exit code of the node

\$PRE_SCRIPT_RETURN: exit code of PRE script

\$RETRY: current retry count

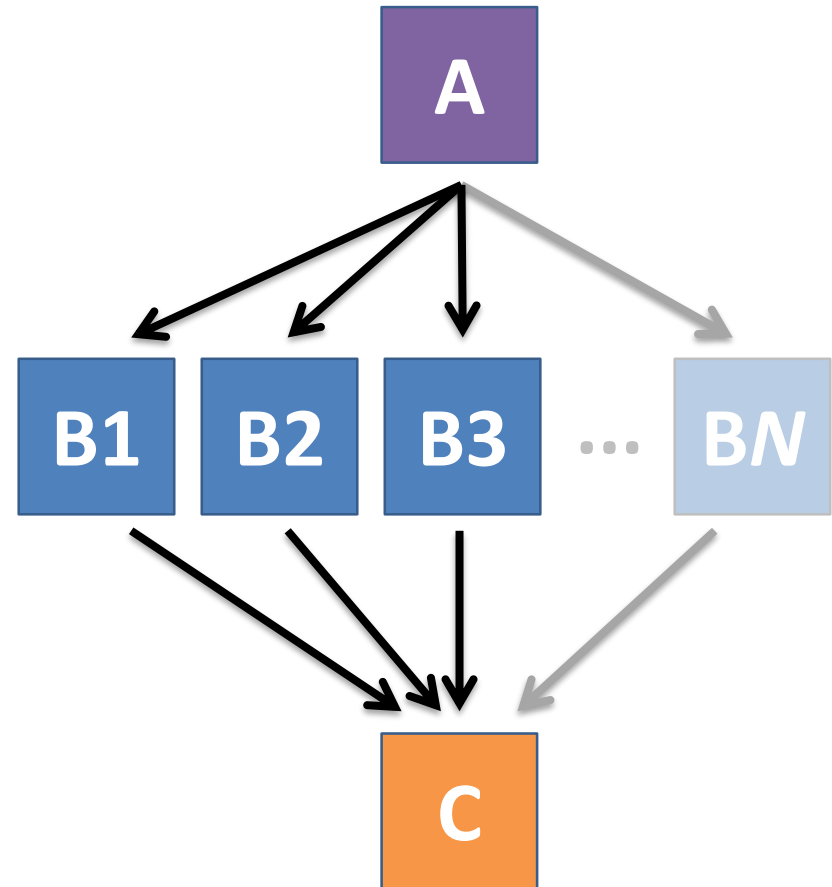
(more variables described in the manual)

[DAGMan Applications > DAG Input File > SCRIPT](#)

[DAGMan Applications > Advanced Features > Retrying](#)

Best Control Achieved with One Process per JOB Node

- While submit files can ‘queue’ many processes, a *single process per submit* file is usually best for DAG JOBS
 - Failure of any process in a JOB node results in failure of the entire node and immediate removal of other processes in the node.
 - RETRY of a JOB node retries the entire submit file.



Modular Organization and Control of DAG Components

Submit File Templates via VARS

- **VARS** line defines node-specific values that are passed into submit file variables

```
VARS node_name var1="value" [var2="value"]
```

- Allows a single submit file shared by all B jobs, rather than one submit file for each JOB.

my.dag

```
JOB B1 B.sub  
VARS B1 data="B1" opt="10"  
JOB B2 B.sub  
VARS B2 data="B2" opt="12"  
JOB B3 B.sub  
VARS B3 data="B3" opt="14"
```

B.sub

```
...  
InitialDir = $(data)  
arguments = $(data).csv $(opt)  
...  
queue
```

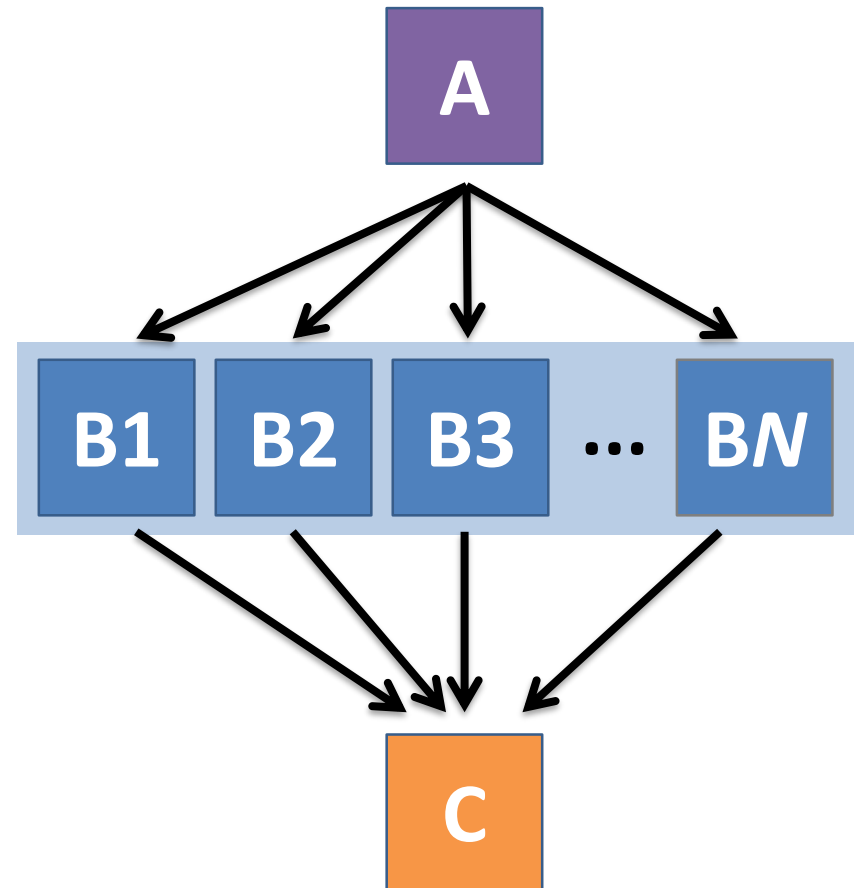

SPLICE groups of nodes to simplify lengthy DAG files

my.dag

```
JOB A A.sub  
SPLICE B B.sp1  
JOB C C.sub  
PARENT A CHILD B  
PARENT B CHILD C
```

B.sp1

```
JOB B1 B1.sub  
JOB B2 B2.sub  
...  
JOB BN BN.sub
```



Use nested SPLICEs with DIR for repeating workflow components

my.dag

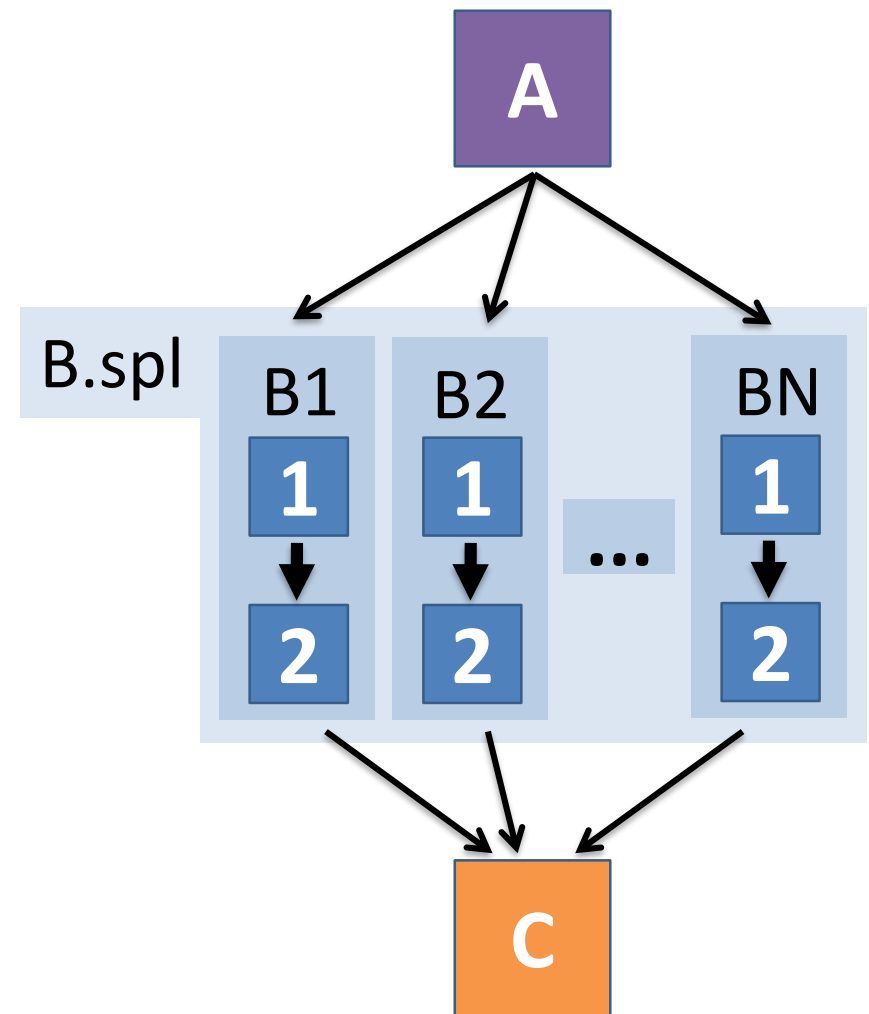
```
JOB A A.sub DIR A
SPLICE B B.spl DIR B
JOB C C.sub DIR C
PARENT A CHILD B
PARENT B CHILD C
```

B.spl

```
SPLICE B1 ../inner.spl DIR B1
SPLICE B2 ../inner.spl DIR B2
...
SPLICE BN ../inner.spl DIR BN
```

inner.spl

```
JOB 1 ../1.sub
JOB 2 ../2.sub
PARENT 1 CHILD 2
```



Use nested SPLICES with DIR for repeating workflow components

my.dag

```
JOB A A.sub DIR A
SPLICE B B.sp1 DIR B
JOB C C.sub DIR C
PARENT A CHILD B
PARENT B CHILD C
```

B.sp1

```
SPLICE B1 ../inner.sp1 DIR B1
SPLICE B2 ../inner.sp1 DIR B2
...
SPLICE BN ../inner.sp1 DIR BN
```

inner.sp1

```
JOB 1 ../1.sub
JOB 2 ../2.sub
PARENT 1 CHILD 2
```

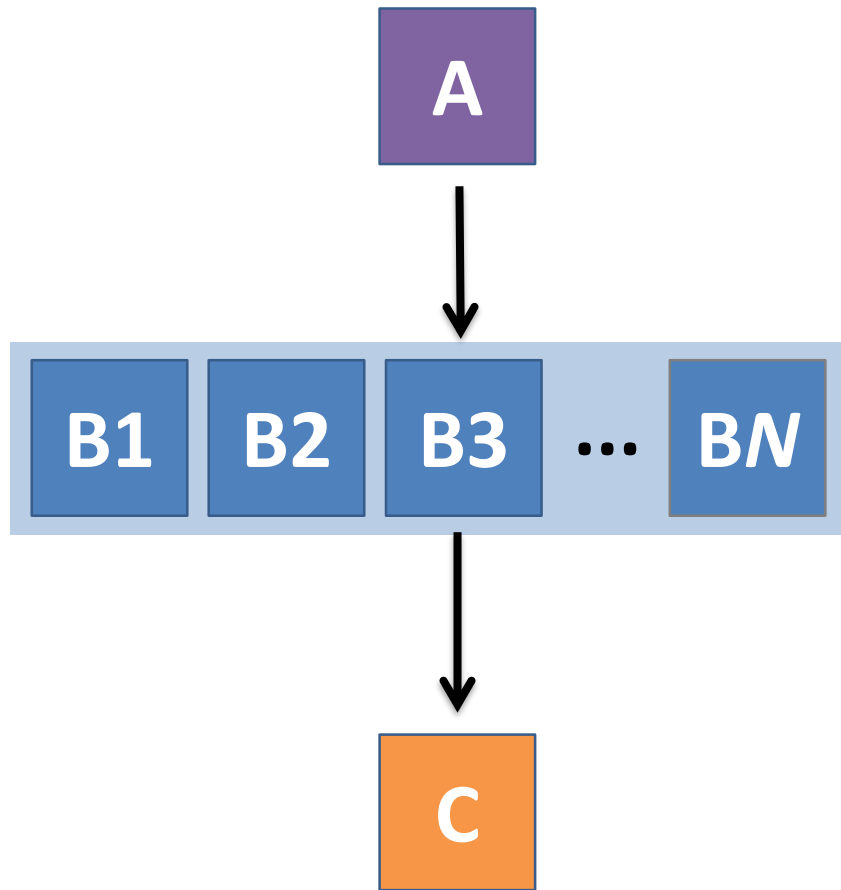
(dag_dir) /

```
my.dag
A/ A.sub      (A job files)
B/ B.sp1     inner.sp1
   1.sub     2.sub
   B1/      (1-2 job files)
   B2/      (1-2 job files)
   ...
   BN/      (1-2 job files)
C/ C.sub      (C job files)
```

More on SPLICE Behavior

- **HTCondor takes in a DAG and its SPLICES as a single, large DAG file.**
 - SPLICES simply allow the user to simplify and modularize the DAG expression using separate files
 - A single DAGMan job is queued with single set of status files.
- Great for gradually testing and building up a large DAG (since a SPLICE file can be submitted by itself, without its outer DAG).
- SPLICE lines are not treated like nodes.
 - no PRE/POST scripts or RETRIES

What if some DAG components can't be known at submit time?



e.g. If the value of ***N*** can only be determined as part of the work of the prior node (**A**) ...

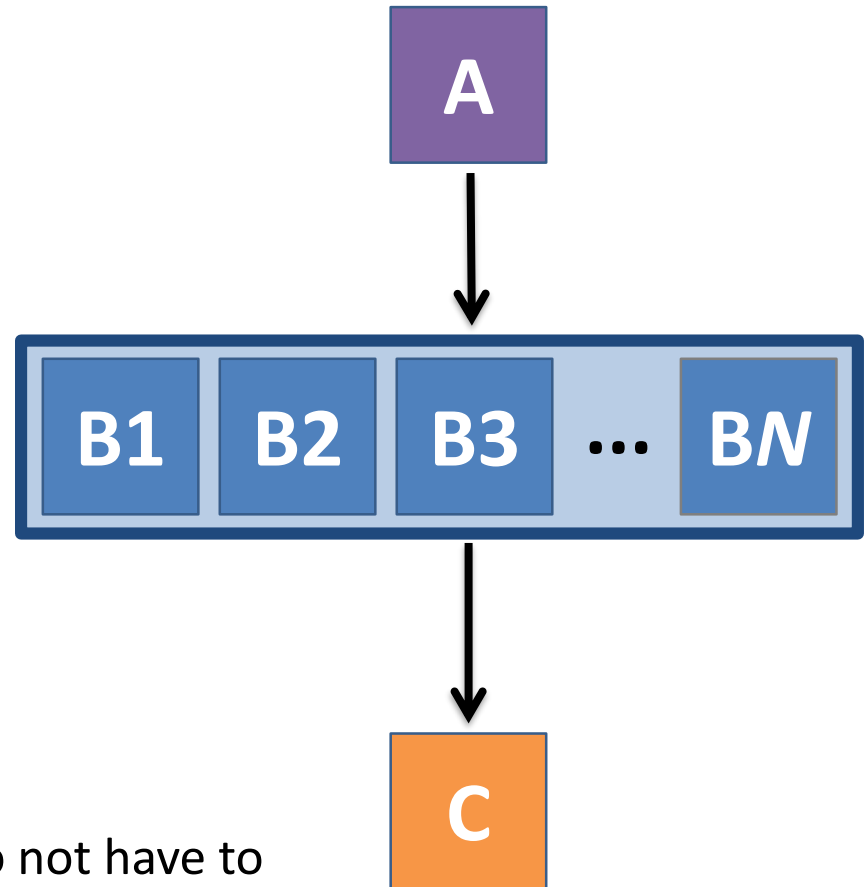
A SUBDAG within a DAG

my.dag

```
JOB A A.sub
SUBDAG EXTERNAL B B.dag
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
```

B.dag (written by **A**)

```
JOB B1 B1.sub
JOB B2 B2.sub
...
JOB BN BN.sub
```



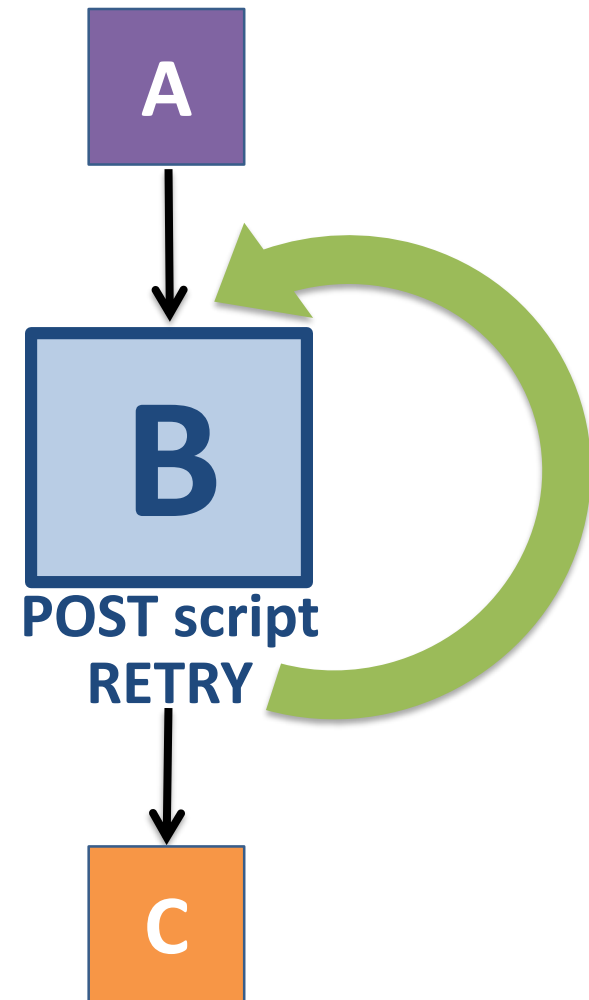
A SUBDAG is not submitted (so contents do not have to exist) until prior nodes in the outer DAG have completed.

Use a SUBDAG to achieve Cyclic Components within a DAG

- POST script determines whether another iteration is necessary; if so, exits non-zero
- RETRY applies to entire SUBDAG, which may include multiple, sequential nodes

my.dag

```
JOB A A.sub
SUBDAG EXTERNAL B B.dag
SCRIPT POST B iterateB.sh
RETRY B 100
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
```



More on SUBDAG Behavior

- Each SUBDAG EXTERNAL is a DAGMan job running in the queue, and too many can overwhelm the queue.
 - **WARNING:** SUBDAGs should only be used (rather than SPLICES) when absolutely necessary!
- SUBDAGs are nodes (can have PRE/POST scripts, retries, etc.)

DAG-level Control

Pause (then resume) a DAG by holding it

- Hold the DAGMan job process:
`condor_hold dagman_jobID`
- Pauses the DAG
 - No new node jobs submitted
 - Queued node jobs continue to run (including SUBDAGs), but no PRE/POST scripts
 - DAG resumes when released
(`condor_release dagman_jobID`)

Cleanly quit a DAG with a halt file

- Create a file named *DAG_file.halt* in the same directory as the submitted DAG file
- Allows the DAG to complete nodes in-progress
 - No new node jobs submitted
 - Queued node jobs, SUBDAGs, and **POST scripts** continue to run, but not PRE scripts
- DAGMan resumes after the file is deleted
 - **If not deleted**, the DAG creates a rescue DAG file and exits after all queued jobs have completed

[DAGMan > Suspending a Running DAG](#)

[DAGMan > The Rescue DAG](#)

Throttle job nodes of large DAGs via DAG-level configuration

- If a DAG has *many* (thousands or more) jobs, submit server and queue performance can be assured by limiting:
 - Number of jobs in the queue
 - Number of jobs idle (waiting to run)
 - Number of PRE or POST scripts running
- Limits can be specified in a DAG-specific **CONFIG** file (recommended) or as arguments to `condor_submit_dag`

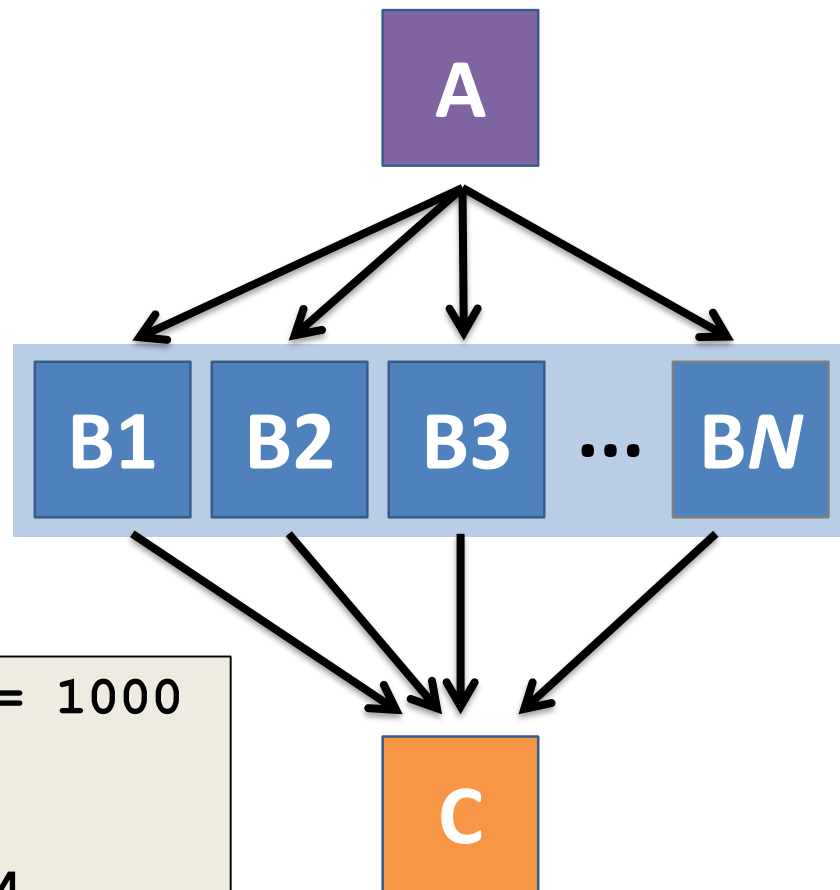
DAG-specific throttling via a CONFIG file

my.dag

```
JOB A A.sub
SPLICE B B.dag
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
CONFIG my.dag.config
```

my.dag.config

```
DAGMAN_MAX_JOBS_SUBMITTED = 1000
DAGMAN_MAX_JOBS_IDLE = 100
DAGMAN_MAX_PRE_SCRIPTS = 4
DAGMAN_MAX_POST_SCRIPTS = 4
```



Other DAGMan Features

Other DAGMan Features: Node-Level Controls

- Set the **PRIORITY** of JOB nodes with:

PRIORITY node_name priority_value

- Use a **PRE_SKIP** to skip a node and mark it as successful, if the PRE script exits with a specific exit code:

PRE_SKIP node_name exit_code

[DAGMan Applications > Advanced Features > Setting Priorities](#)

[DAGMan Applications > The DAG Input File > PRE_SKIP](#)

Other DAGMan Features: Modular Control

- Append **NOOP** to a JOB definition so that its JOB process isn't run by DAGMan
 - Test DAG structure without running jobs (node-level)
 - Simplify combinatorial PARENT-CHILD statements (modular)
- Communicate DAG features separately with **INCLUDE**
 - e.g. separate file for JOB nodes and for VARS definitions, as part of the same DAG
- Define a **CATEGORY** to throttle only a specific subset of jobs

[DAGMan Applications > The DAG Input File > JOB](#)

[DAGMan Applications > Advanced Features > INCLUDE](#)

[DAGMan Applications > Advanced > Throttling by Category](#)

Other DAGMan Features: DAG-Level Controls

- Replace the *node_name* with **ALL_NODES** to apply a DAG feature to all nodes of the DAG
- Abort the entire DAG if a specific node exits with a specific exit code:

ABORT-DAG-ON *node_name* *exit_code*

- Define a **FINAL** node that will always run, even in the event of DAG failure (to clean up, perhaps).

FINAL *node_name* *submit_file*

[DAGMan Applications > Advanced > ALL_NODES](#)

[DAGMan Applications > Advanced > Stopping the Entire DAG](#)

[DAGMan Applications > Advanced > FINAL Node](#)

Much More in the HTCCondor Manual!!!

http://research.cs.wisc.edu/htcondor/manual/current/2_10DAGMan_Applications.html



FINAL QUESTIONS?

htcondor-users@cs.wisc.edu