AN INTRODUCTION TO WORKFLOWS WITH DAGMAN

Presented by Lauren Michael
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 replicate the same workflow multiple times in the future.
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

wikipedia.org/wiki/Directed_acyclic_graph
Describing Workflows with DAGMan
DAGMan in the HTCondor Manual

- 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 will communicate the “nodes” and directional “edges” of the DAG
Simple Example for this Tutorial

- The DAG input file will communicate 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**

```plaintext
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

[Diagram of workflow processes with various tasks and nodes labeled]

Wikimedia Commons

https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator
Endless Workflow Possibilities
Repeating DAG Components!!

[Diagram of repeating DAG components]

https://confluence.pegasus.isi.edu/display/pegasus/LIGO+IHOPE
DAGs are also useful for non-sequential work

‘bag’ of HTC jobs

| B1 | B2 | B3 | ... | BN |

disjointed workflows

A → B → D → E
D → C → F
D → G → H
D → I
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:

```bash
$ condor_submit_dag my.dag
```

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

Submitting job(s).
1 job(s) submitted to cluster 87274940.
A submitted DAG creates and DAGMan job process in the queue

- DAGMan runs on the submit server, as a job in the queue

- At first:

```bash
$ 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   _     _     _      _      _     0.0
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

```bash
$ 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 8:08    1     _     3      5   129.0...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 18:18   0+00:00:00   I   0   0.3  B_run.sh
131.0 alice  4/30 18:18   0+00:00:00   I   0   0.3  B_run.sh
132.0 alice  4/30 18:18   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

```bash
$ 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 8:08 4 _ 1 5 129.0...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)/

<table>
<thead>
<tr>
<th>A.sub</th>
<th>B1.sub</th>
<th>B2.sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3.sub</td>
<td>C.sub</td>
<td>(other job files)</td>
</tr>
<tr>
<td>my.dag</td>
<td><strong>my.dag.condor.sub</strong></td>
<td>my.dag.dagman.log</td>
</tr>
<tr>
<td><strong>my.dag.dagman.out</strong></td>
<td>my.dag.lib.err</td>
<td>my.dag.lib.out</td>
</tr>
<tr>
<td>my.dag.nodes.log</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**.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
  ```

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

```
$ 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 8:08 4 _ 1 6 129.0...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
```
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 \texttt{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 when:
  – a node fails, and after DAGMan advances through any other possible nodes
  – the DAG is removed from the queue
     (or aborted; covered later)
  – the DAG is halted and not unhalted
     (covered later)

• Resubmission uses the rescue file (if it exists) when the original DAG file is resubmitted
  – override: `condor_submit_dag dag_file -f`
Node Failures Result in DAG Failure

- 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

DAGMan > The Rescue DAG
Resolving held node jobs

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)/

<table>
<thead>
<tr>
<th>A.sub</th>
<th>B1.sub</th>
<th>B2.sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3.sub</td>
<td>C.sub</td>
<td>(other job files)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>my.dag</th>
<th>my.dag.condor.sub</th>
<th>my.dag.dagman.log</th>
</tr>
</thead>
<tbody>
<tr>
<td>my.dag.dagman.out</td>
<td>my.dag.lib.err</td>
<td>my.dag.lib.out</td>
</tr>
<tr>
<td>my.dag.nodes.log</td>
<td>my.dag.dagman.metrics</td>
<td></td>
</tr>
</tbody>
</table>

*.dagman.metrics is a summary of events and outcomes
*.dagman.log will note the completion of the DAGMan job
*.dagman.out has detailed logging (look to first for errors)
Beyond the Basic DAG: Node-level Modifiers
Default File Organization

my.dag

<table>
<thead>
<tr>
<th>JOB</th>
<th>A</th>
<th>A.sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB</td>
<td>B1</td>
<td>B1.sub</td>
</tr>
<tr>
<td>JOB</td>
<td>B2</td>
<td>B2.sub</td>
</tr>
<tr>
<td>JOB</td>
<td>B3</td>
<td>B3.sub</td>
</tr>
<tr>
<td>JOB</td>
<td>C</td>
<td>C.sub</td>
</tr>
<tr>
<td>PARENT</td>
<td>A</td>
<td>CHILD</td>
</tr>
<tr>
<td>PARENT</td>
<td>B1</td>
<td>B2</td>
</tr>
</tbody>
</table>

(dag_dir)/

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A.sub</td>
<td>B1.sub</td>
<td></td>
</tr>
<tr>
<td>B2.sub</td>
<td>B3.sub</td>
<td></td>
</tr>
<tr>
<td>C.sub</td>
<td>my.dag</td>
<td></td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>JOB</th>
<th>File</th>
<th>DIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A.sub</td>
<td>A</td>
</tr>
<tr>
<td>B1</td>
<td>B1.sub</td>
<td>B</td>
</tr>
<tr>
<td>B2</td>
<td>B2.sub</td>
<td>B</td>
</tr>
<tr>
<td>B3</td>
<td>B3.sub</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C.sub</td>
<td>C</td>
</tr>
</tbody>
</table>

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 jobs:

```
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 the 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]), or retry scripts ([DEFER])

- **Note:** Unnecessary for nodes (jobs) that can use `max_retries` in the submit file
**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

$JOB$: 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)*

```
JOB A A.sub
SCRIPT POST A checkA.sh my.out $RETURN
RETRY A 5
```
Best Control Achieved with One Process per **JOB** Node

- While submit files can ‘queue’ many processes, a **single process per submit** file is best for DAG JOBs
  - Failure of any process in a JOB node results on 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 \textbf{VARS}

- \textbf{VARS} line defines node-specific values that are passed into submit file variables
  \texttt{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.

\begin{verbatim}
my.dag
\end{verbatim}

\begin{verbatim}
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”
\end{verbatim}

\begin{verbatim}
B.sub
\end{verbatim}

\begin{verbatim}
... 
InitialDir = $(data)
arguments = $(data).csv $(opt)
... 
queue
\end{verbatim}
**SPLICE** groups of nodes to simplify lengthy DAG files

**my.dag**

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

**B.spl**

```
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

<table>
<thead>
<tr>
<th>JOB</th>
<th>A</th>
<th>A.sub</th>
<th>DIR A</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLICE</td>
<td>B</td>
<td>B.spl</td>
<td>DIR B</td>
</tr>
<tr>
<td>JOB</td>
<td>C</td>
<td>C.sub</td>
<td>DIR C</td>
</tr>
<tr>
<td>PARENT</td>
<td>A</td>
<td>CHILD</td>
<td>B</td>
</tr>
<tr>
<td>PARENT</td>
<td>B</td>
<td>CHILD</td>
<td>C</td>
</tr>
</tbody>
</table>
```

```
B.spl

<table>
<thead>
<tr>
<th>SPLICE</th>
<th>B1</th>
<th>../inner.spl</th>
<th>DIR B1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLICE</td>
<td>B2</td>
<td>../inner.spl</td>
<td>DIR B2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLICE</td>
<td>BN</td>
<td>../inner.spl</td>
<td>DIR BN</td>
</tr>
</tbody>
</table>
```

```
inner.spl

<table>
<thead>
<tr>
<th>JOB</th>
<th>1</th>
<th>../1.sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB</td>
<td>2</td>
<td>../2.sub</td>
</tr>
<tr>
<td>PARENT</td>
<td>1</td>
<td>CHILD</td>
</tr>
</tbody>
</table>
```

```
(dag_dir)/

my.dag

<table>
<thead>
<tr>
<th>A/</th>
<th>A.sub</th>
<th>(A job files)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B/</td>
<td>B.spl</td>
<td>inner.spl</td>
</tr>
<tr>
<td></td>
<td>1.sub</td>
<td>2.sub</td>
</tr>
<tr>
<td></td>
<td>B1/</td>
<td>(1-2 job files)</td>
</tr>
<tr>
<td></td>
<td>B2/</td>
<td>(1-2 job files)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BN/</td>
<td>(1-2 job files)</td>
</tr>
<tr>
<td>C/</td>
<td>C.sub</td>
<td>(C job files)</td>
</tr>
</tbody>
</table>
```
More on **SPLICE** Behavior

- Upon submission of the outer DAG, nodes in the SPLICE(s) are added by DAGMan into the overall DAG structure.
  - 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, as a complete DAG).
- SPLICE lines are not treated like nodes.
  - no PRE/POST scripts or RETRIES (though this may change)
What if some DAG components can’t be known at submit time?

If $N$ can only be determined as part of the work of $A$ …
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
More on **SUBDAG** Behavior

- **WARNING:** SUBDAGs should only be used (over SPLICES) when absolutely necessary!
  - Each SUBDAG EXTERNAL has it’s own DAGMan job running in the queue.
- SUBDAGs are nodes (can have PRE/POST scripts, retries, etc.)
- A SUBDAG is not submitted 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

```plaintext
my.dag

JOB A A.sub
SUBDAG EXTERNAL B B.dag
SCRIPT POST B iterateB.sh
RETRY B 1000
JOB C C.sub
PARENT A CHILD B
PARENT B CHILD C
```
DAG-level Control
Pause a running DAG with hold/release

• Hold the DAGMan job process:
  \texttt{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
  – DAGMan jobs remains in the queue until released (\texttt{condor\_release}) or removed
Pause a DAG with a **halt file**

- Create a file named `DAG_file.halt` in the same directory as the submitted DAG file
- Pauses the DAG
  - 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 rescue DAG file and exits after all queued jobs have completed
Throttle job nodes of large DAGs via DAG-level configuration

• If a DAG has *many* (thousands or more) jobs, performance of the submit server and queue 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**

```plaintext
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**

```plaintext
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`
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
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 HTCondor Manual!!!

FINAL QUESTIONS?

htcondor-users@cs.wisc.edu