HTCondor and Workflows: Tutorial

HTCondor Week 2016

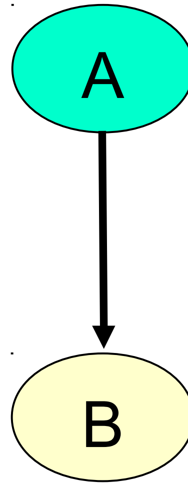
Kent Wenger

What are workflows?

- General: a sequence of connected steps
- Our case:
  - Steps are HTCondor jobs (really, submit files)
  - Sequence defined at higher level
  - Controlled by a Workflow Management System (WMS), *not just a script*

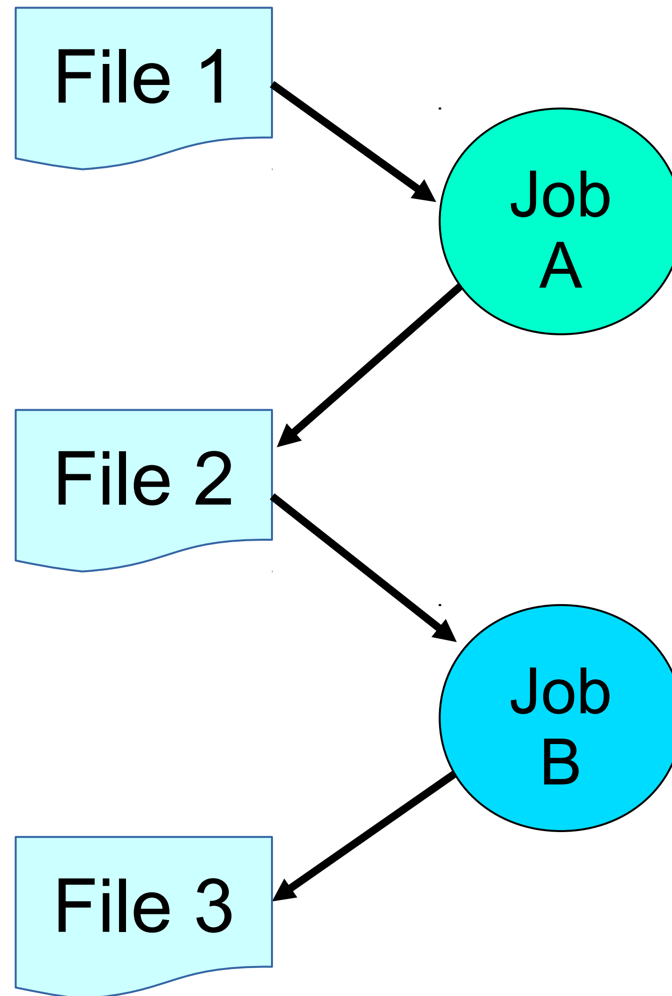
Why use workflows?

- Dependencies between jobs:



- Job A must complete successfully before job B starts
- Typically, job A produces output file(s) that are inputs to job B
- Can't do this with a single submit file

Workflow example with files



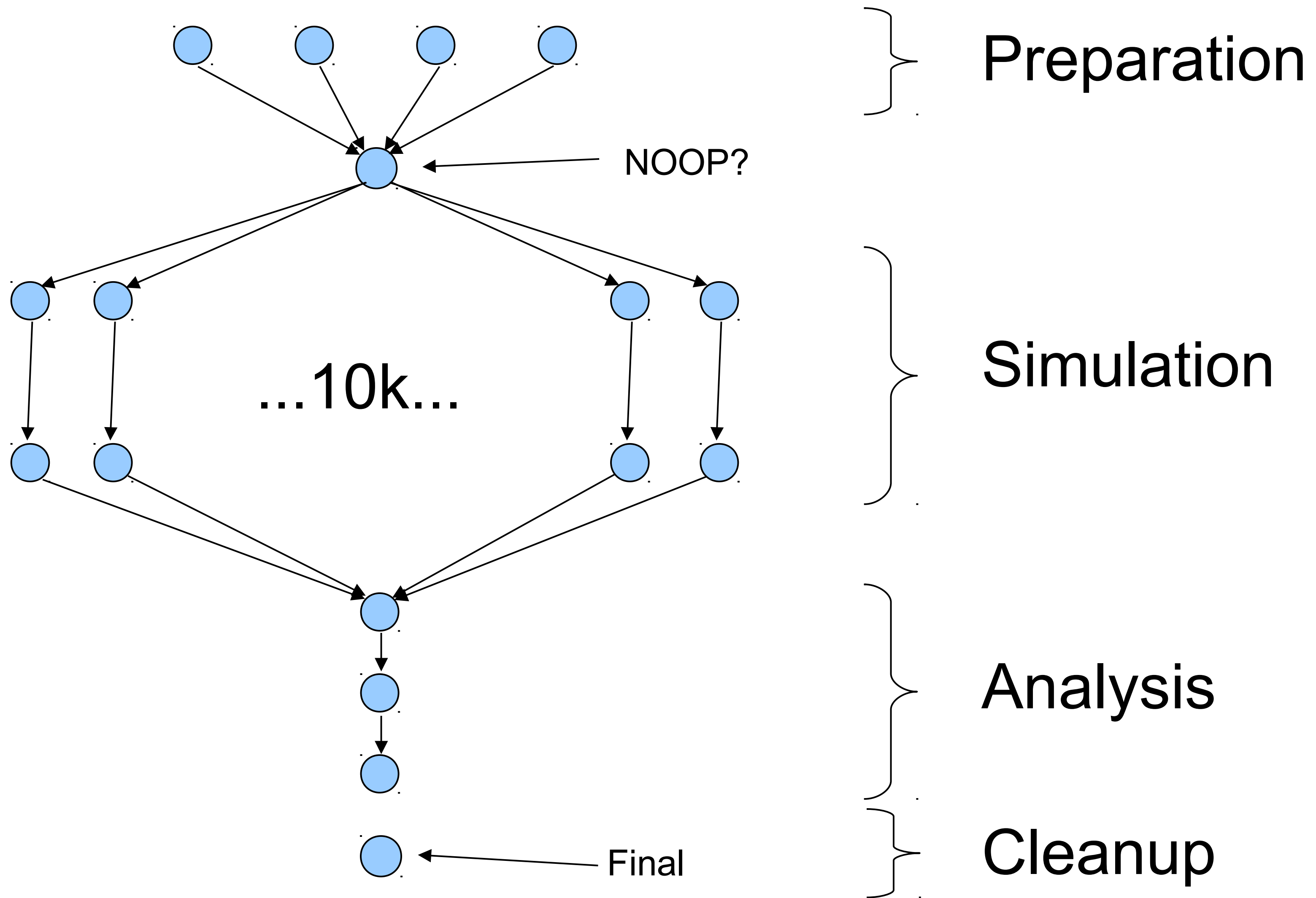
Workflows – launch and forget

- Automates tasks user *could* perform manually (for example, the previous slide)…
  - But Workflow Management System takes care of automatically
- A workflow can take days, weeks or even months
- The result: one user action can utilize many resources while maintaining complex job inter-dependencies and data flows

Workflow management systems

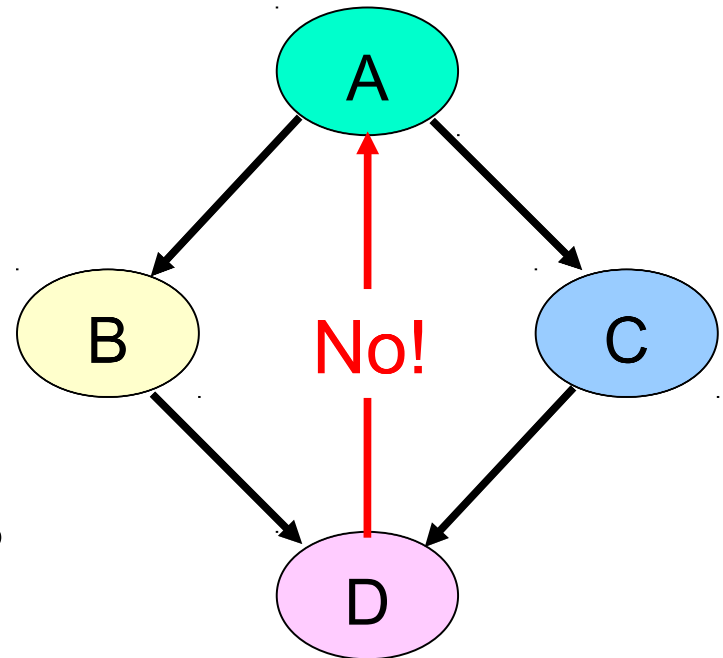
- DAGMan (Directed Acyclic Graph Manager)
    - HTCondor's WMS (this talk)
- Pegasus
    - A higher level on top of DAGMan
        - Data- and grid-aware
    - Hands-on tutorial this afternoon (separate session)
    - Talk Friday

Example workflow



DAG (directed acyclic graph) definitions

- DAGs have one or more nodes (or vertices)
- Dependencies are represented by arcs (or edges). These are arrows that go from parent to child)
- No cycles!



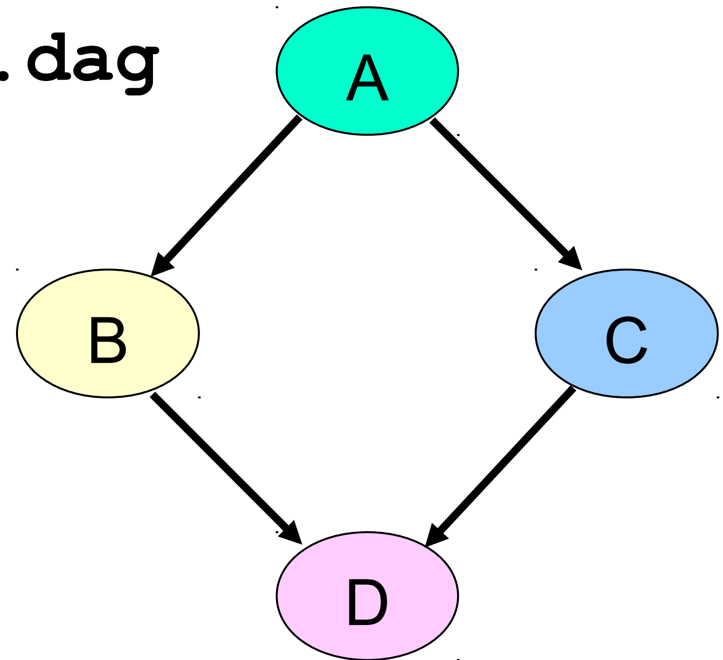
Basic DAG commands

- Job command defines a name, associates that name with an HTCondor submit file
  - The name is used in many other DAG commands
  - Required in all DAG files
  - “Job” should really be “node”
- Parent...child command creates a dependency between nodes
  - Child cannot run until parent completes successfully

Defining a DAG to DAGMan

A DAG input file defines a DAG:

```
# file name: diamond.dag
Job A a.submit
Job B b.submit
Job C c.submit
Job D d.submit
Parent A Child B C
Parent B C Child D
```

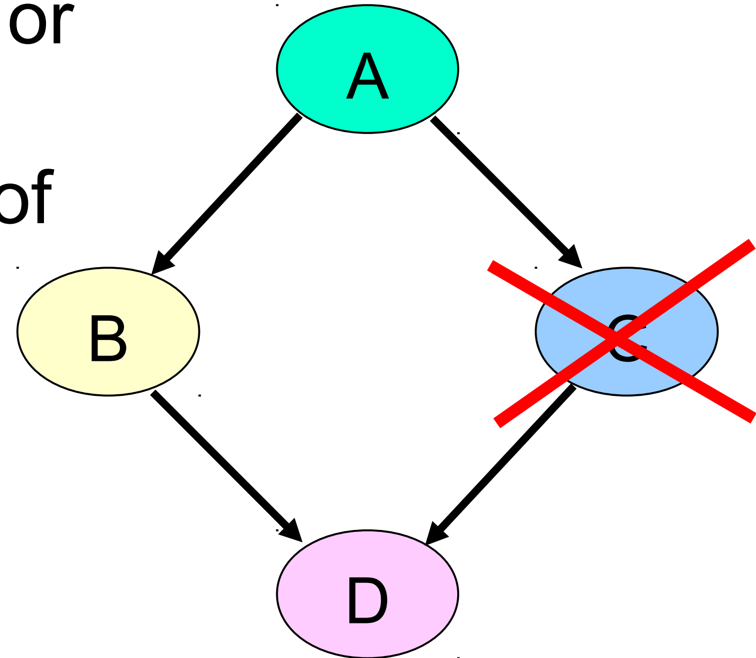


Jobs/clusters

- Submit description files used in a DAG can create multiple jobs, but they must all be in a single cluster.
  - A submit file that creates >1 cluster causes node failure.
- The failure of any job means the entire cluster fails. Other jobs in the cluster are removed.
- Don't use large clusters within DAGs.

Node success or failure

- A node either succeeds or fails
- Based on the exit code of the job(s)
  - 0: success
  - not 0: failure
- This example: C fails
- Failed nodes block execution; DAG fails



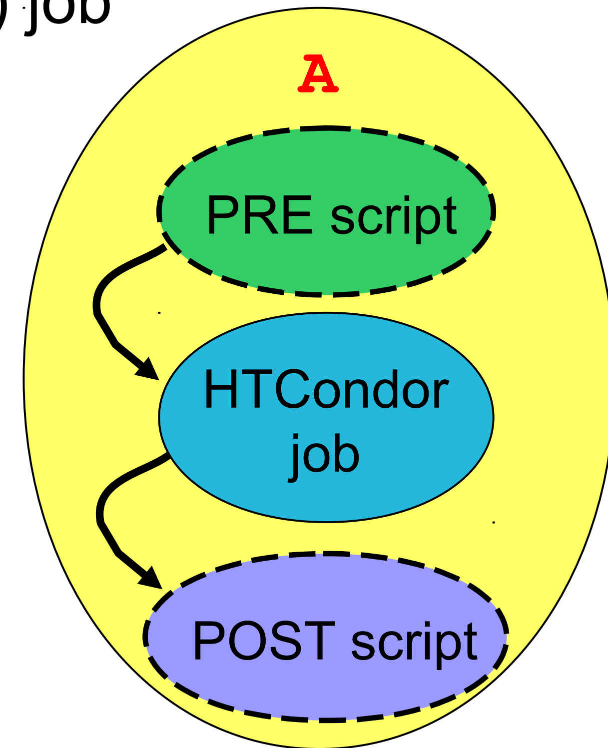
PRE/POST scripts – why?

- Set up input
- Check output
- Dynamically create submit file or sub-DAG (more later)
- Probably lots of other reasons…

- Should be lightweight (run on submit machine)

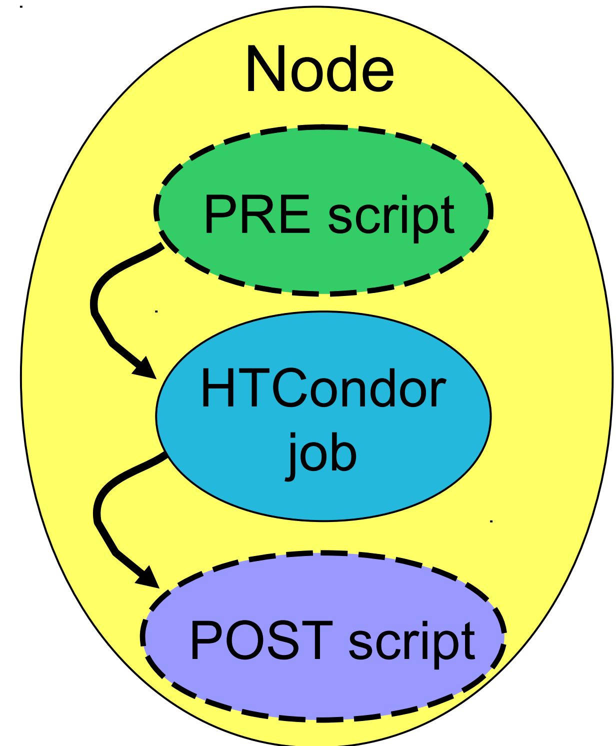
PRE and POST scripts (cont)

- DAGMan allows optional PRE and/or POST scripts for any node
  - Not necessarily a script: any executable
  - Run before (PRE) or after (POST) job
  - Scripts run on submit machine (not execute machine)
- In the DAG input file:
  - `Job A a.submit`
  - `SCRIPT PRE A script_name arguments`
  - `SCRIPT POST A script_name arguments`



DAG node with scripts

- DAGMan treats the node as a unit (e.g., dependencies are between nodes)
- PRE script, Job, or POST script determines node success or failure (table in manual gives details)
- If PRE script fails, job and POST script are not run (changed as of 8.5.5)
- If job fails, POST *is* run



Script argument variables

- **`$JOB`**: node name
- **`$JOBID`**: Condor ID (*cluster.proc*) (POST only)
- **`$RETRY`**: current retry
- **`$MAX_RETRIES`**: max # of retries
- **`$RETURN`**: exit code of HTCondor job (POST only)
- **`$PRE_SCRIPT_RETURN`**: PRE script return value (POST only)
- **`$DAG_STATUS`**: A number indicating the state of DAGMan. See the manual for details.
- **`$FAILED_COUNT`**: the number of nodes that have failed in the DAG

Don't re-do work: PRE_SKIP

- Allows PRE script to immediately declare node successful (job and POST script are not run)
- In the DAG input file:

```
JOB A A.cmd
SCRIPT PRE A A.pre
PRE_SKIP node_name non-zero_integer
```

- If the PRE script of A exits with the specified value, the node succeeds immediately, and the node job and POST script are skipped.
- If PRE script succeeds, node job and POST are run.
- If the PRE script fails with a different value, the node job and POST script are skipped (as if PRE_SKIP were not specified).

Submitting a DAG to HTCondor

- To submit an entire DAG, run

    `condor_submit_dag` ***DagFile***

- **`condor_submit_dag`** creates a submit description file for DAGMan, and DAGMan itself is submitted as an HTCondor job (in the scheduler universe)
- **`-f(orce)`** option forces overwriting of existing files (to re-run a previously-run DAG)
- Don't try to run duplicate DAG instances!

Monitoring running DAGs: condor_q -dag

- Shows current workflow state
- The **-dag** option associates DAG node jobs with the parent DAGMan job

```
> condor_q -dag
-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9618?...
 ID      OWNER/NODENAME       SUBMITTED      RUN_TIME ST PRI SIZE CMD
 767.0   wenger              5/15 10:41   0+00:00:32 R  0    2.2 condor_dagman -
 768.0     |-sub01           5/15 10:41   0+00:00:26 R  0    2.2 condor_dagman -
 771.0      |-nodeS01        5/15 10:41   0+00:00:12 R  0    0.0 sleep 30
 772.0      |-nodeS02        5/15 10:41   0+00:00:12 R  0    0.0 sleep 30
 769.0     |-node01          5/15 10:41   0+00:00:12 R  0    0.0 sleep 30
 770.0     |-node02          5/15 10:41   0+00:00:12 R  0    0.0 sleep 30

6 jobs; 0 completed, 0 removed, 0 idle, 6 running, 0 held, 0 suspended
```

Monitoring running DAGS: condor_q -batch

- A single line of output summarizing all jobs with the same batch name:

```
> condor_q -batch
-- Schedd: wenger@manta.cs.wisc.edu : <128.105.14.228:9618?...
OWNER   BATCH_NAME      SUBMITTED    DONE    RUN     IDLE   TOTAL JOB_IDS
wenger tmp.dag+779   5/15 10:45      3       2       1      12 784.0 ... 786.0

5 jobs; 0 completed, 0 removed, 1 idle, 4 running, 0 held, 0 suspended
```

Monitoring a DAG: dagman.out file

- Logs detailed workflow history
- Mostly for debugging – first place to look if something goes wrong!
- ***`DagFile.dagman.out`***
- Verbosity controlled by the **`DAGMAN_VERBOSITY`** configuration macro and **`-debug`** ***`n`*** on the **`condor_submit_dag`** command line
  - 0: least verbose
  - 7: most verbose
- Don't decrease verbosity unless really necessary

Dagman.out contents

```
...
04/17/11 13:11:26 Submitting Condor Node A job(s)...
04/17/11 13:11:26 submitting: condor_submit -a dag_node_name' '=' 'A -a +DAGManJobId' '='
    '180223 -a DAGManJobId' '=' '180223 -a submit_event_notes' '=' 'DAG' 'Node:' 'A -a
    +DAGParentNodeNames' '=' '"" dag_files/A2.submit
04/17/11 13:11:27 From submit: Submitting job(s).
04/17/11 13:11:27 From submit: 1 job(s) submitted to cluster 180224.
04/17/11 13:11:27       assigned Condor ID (180224.0.0)
04/17/11 13:11:27 Just submitted 1 job this cycle...
04/17/11 13:11:27 Currently monitoring 1 Condor log file(s)
04/17/11 13:11:27 Event: ULOG_SUBMIT for Condor Node A (180224.0.0)
04/17/11 13:11:27 Number of idle job procs: 1
04/17/11 13:11:27 Of 4 nodes total:
04/17/11 13:11:27  Done     Pre   Queued    Post   Ready   Un-Ready   Failed
04/17/11 13:11:27   ===     ===      ===     ===     ===        ===      ===
04/17/11 13:11:27     0       0        1       0       0          3        0
04/17/11 13:11:27 0 job proc(s) currently held
...
```

This is a small excerpt of the dagman.out file.

Removing a running DAGs: condor_rm

- `condor_rm` *`dagman_id`*
  - Removes *entire* workflow
    - Removes all queued node jobs
    - Kills PRE/POST scripts
  - Creates rescue DAG (more on this on later)
  - Work done by partially-completed node jobs is lost
    - Relatively small jobs are good

Pausing a running DAG: hold/release

- **`condor_hold` *`dagman_id`***
  - "Pauses" the DAG
    - Queued node jobs continue
    - No new node jobs submitted
    - No PRE or POST scripts are run
  - DAGMan stays in queue if not released
- **`condor_release` *`dagman_id`***
  - DAGMan "catches up", starts submitting jobs

Pausing a running DAG: halt file

- "Pauses" the DAG (different semantics than hold)
  - Queued node jobs continue
  - POST scripts are run as jobs finish
  - No new jobs will be submitted and no PRE scripts will be run
- When all submitted jobs complete, DAGMan creates a rescue DAG and exits (if not un-halted)

Halting a DAG (cont)

- Create a file named ***`DagFile.halt`*** in the same directory as your DAG file.
- Remove halt file to resume normal operation
- Should be noticed w/in 5 sec (**`DAGMAN_USER_LOG_SCAN_INTERVAL`**)

- Good if load on submit machine is very high
- Avoids hold/release problem of possible duplicate PRE/POST script instances

Make warnings into errors: DAGMAN_USE_STRICT

- Warnings are printed to **`dagman.out`** file – easy to ignore
- **`DAGMAN_USE_STRICT`** turns warnings into fatal errors
  - Example: node category has no assigned nodes
- 0: no warnings become errors
- 1: severe warnings become errors
- 2: medium-severity warnings become errors
- 3: almost all warnings become errors
- Default is 1 (a good idea to increase)

Handling failures: node retries

- For possibly transient errors
- Before a node is considered failed. . .
  - Retry N times. In the DAG file:

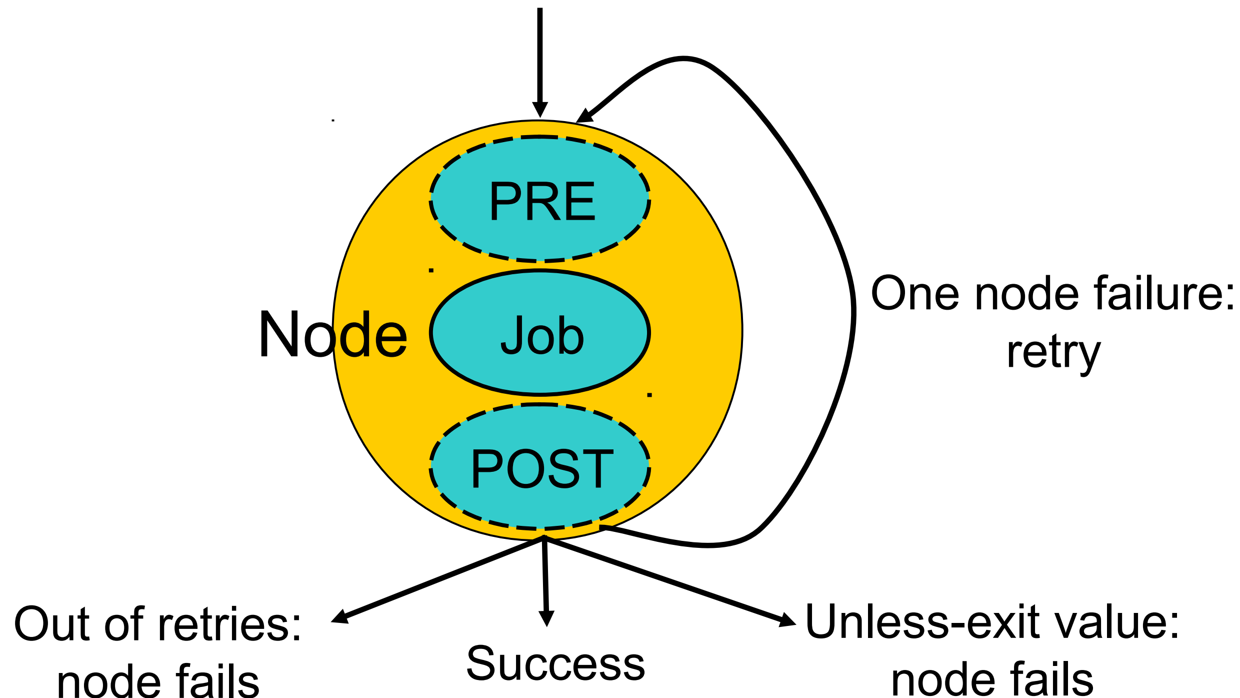
    RETRY *node_name max_retries*
  - Example: **`RETRY C 4`**

    (to retry node **`C`** four times before calling the node failed)
  - Retry N times, unless a node returns specific exit code. In the DAG file:

    RETRY *node_name max_retries* UNLESS-EXIT *exit_code*

Node retries, continued

- Node is retried as a whole



Handling failures: script deferral

- Re-try failed script (not entire node) *after a specified deferral time*
- Deferred scripts don't count against maxpre/maxpost
- In the DAG file:

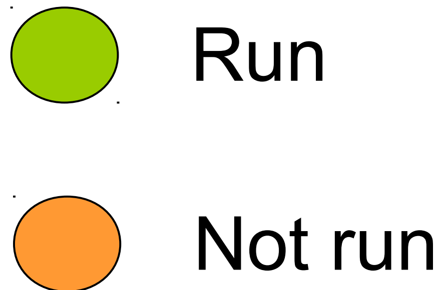
```
SCRIPT [DEFER status time] PRE|POST
  node script_path args...
```

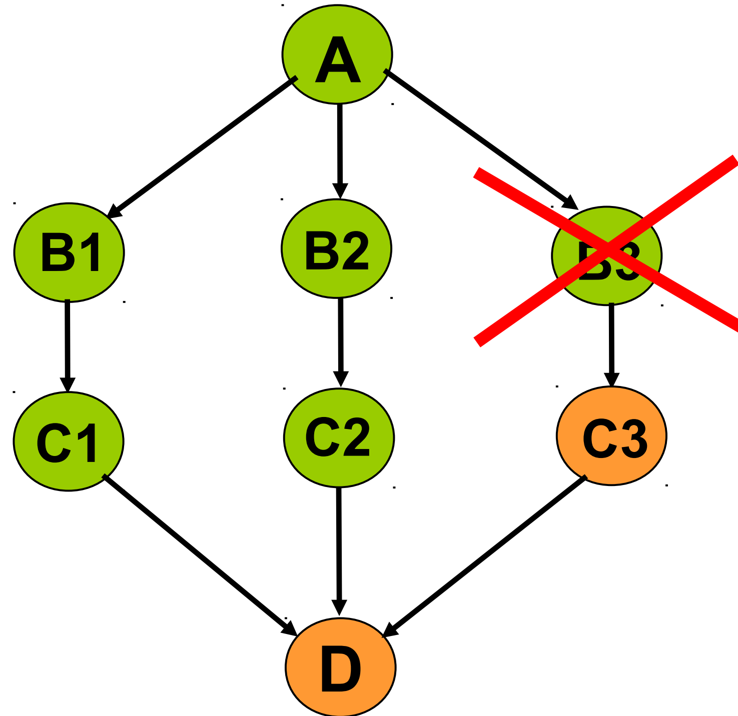
- If script exits with ***status***, re-try after ***time*** (or more) seconds
- Added in 8.3.5

Handling failures: Rescue DAGs

- Save the state of a partially-completed DAG
- Created when a node fails (after maximal progress) or the **`condor_dagman`** job is removed with **`condor_rm`** or when DAG is halted and all queued node jobs finish or when DAG is aborted
  - DAGMan makes as much progress as possible in the face of failed nodes
- DAGMan immediately exits after writing a rescue DAG file
- Automatically run when you re-submit the original DAG (unless **`–force`** is passed to **`condor_submit_dag`**)

Rescue DAGs (cont)





Rescue DAGs (cont)

- The Rescue DAG file, by default, is only a partial DAG file.
- A partial Rescue DAG file contains only information about which nodes are done, and the number of retries remaining for nodes with retries.
- Does not contain information such as the actual DAG structure and the specification of the submit file for each node job.
- Partial Rescue DAGs are automatically parsed in combination with the original DAG file, which contains information such as the DAG structure.

Rescue DAGs (cont)

- If you change something in the original DAG file, such as changing the submit file for a node job, that change will take effect when running a partial rescue DAG.

Rescue DAG naming

- ***`DagFile.rescue001`***, ***`DagFile.rescue002`***, etc.
- Up to 100 by default (last is overwritten once you hit the limit)
- Newest (highest number) is run automatically when you re-submit the original ***`DagFile`***
- **`condor_submit_dag -dorescuefrom`** ***`number`*** to run specific rescue DAG
  - Newer rescue DAGs are renamed

Composing workflows: sub-DAGs and splices

- Incorporate multiple DAG files into a single workflow
- Sub-DAGs: separate DAGMan instance for each component
- Splices: components are directly incorporated into top-level DAG

Why sub-DAGs?

- Dynamic workflow generation (sub-DAGs can be created "on the fly")
- Re-try multiple nodes as a unit
- Short-circuit parts of the workflow (`ABORT-DAG-ON` in sub-DAG)
- Scalability (can reduce memory footprint)
- Can have different config settings for components

Why splices?

- Advantages of splices over sub-DAGs:
  - Reduced overhead (single DAGMan instance)
  - Simplicity (e.g., single rescue DAG)
  - Throttles apply across entire workflow
  - Unified status for entire workflow (condor_q, etc.)
- Limitations of splices:
  - Splices cannot have PRE and POST scripts (for now)
  - Splices cannot have retries
  - Splice DAGs must exist at submit time

Sub-DAGs

- Multiple DAG files in a single workflow (runs the sub-DAG as a job within the top-level DAG)
- In the DAG input file:
  SUBDAG EXTERNAL ***JobName DagFileName***
- Any number of levels
- Sub-DAG nodes are like any other (can have PRE/POST scripts, retries, DIR, etc.)
- Each sub-DAG has its own DAGMan
  - Separate throttles for each sub-DAG
  - Separate rescue DAGs (run automatically)

Splices

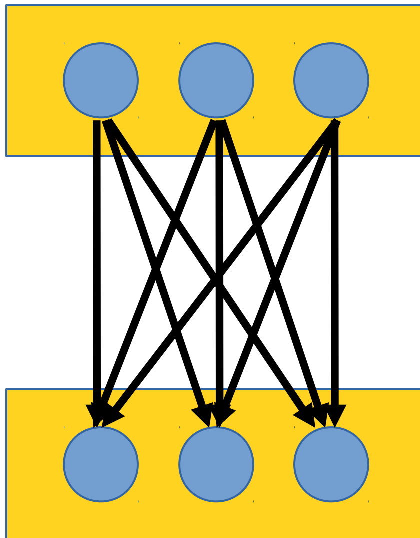
- Multiple DAG files in a single workflow (directly includes splice DAG's nodes within the top-level DAG)
- In the DAG input file:
  SPLICE ***JobName DagFileName***
- Splices can be nested (and combined with sub-DAGs)

Splice pin connections

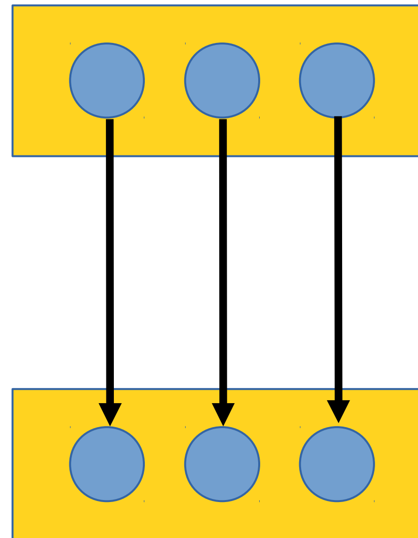
- Ready for beta test (any volunteers?)
- Allows more flexible parent/child relationships between nodes within splices
- Parsed when DAGMan starts up
- *Not* for sub-DAGs

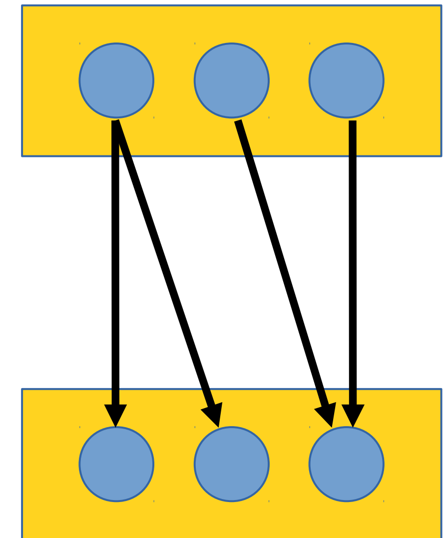
Splice pin connections (cont)

Parent/child



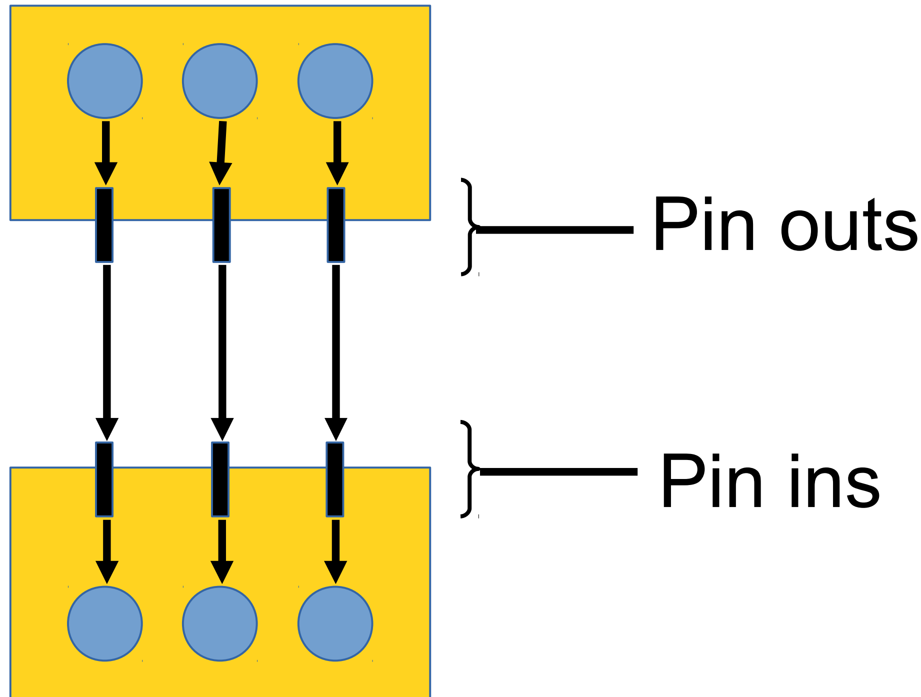
Pin connection





Splice pin connections (cont)

- Define node/pin connections w/in splices
- Pins are connected one-to-one



Splice pin connections (cont)

- Syntax in upper-level DAG:

```
SPLICE splice1_name dag_file1
SPLICE splice2_name dag_file2
CONNECT splice1_name splice2_name
```

- Syntax within splice DAG:

```
PIN_IN node_name pin_number
PIN_OUT node_name pin_number
```

- Pin/node connections can be many-to-many
- Pin numbers start at 1
- Pin outs of splice 1 connected to pin ins of splice 2

Include

- Ready for beta test (any volunteers?)
- Directly incorporates the commands of the specified file
- Parsed when DAGMan starts up
- Syntax:

```
INCLUDE dag_file
```

- Can be used to define **`PIN_IN/PIN_OUT`**

Identifying your workflow: batch name

- Propagated to all parts of a workflow (8.5.5)
  - `JobBatchName` attribute in ClassAds
- Defaults to *`dag_file+cluster`* if not specified (8.5.5)
- Groups jobs in `condor_q` output
- Syntax:

```
condor_submit_dag -batch-name name
   ...
```

New POST script semantics

- POST script is no longer run if PRE script fails (as of 8.5.5)
- Get old semantics by setting **`DAGMAN_ALWAYS_RUN_POST`** to **`True`**

Set ClassAd attributes in DAG file

- Sets attribute in DAGMan's own ClassAd
- Syntax:

```
SET_JOB_ATTR attribute_name = value
```

Don't overload things: throttling

- Limit load on submit machine and pool
  - **`Maxjobs` *N*** limits jobs in queue
  - **`Maxidle` *N*** submit jobs until idle limit is hit
    - Can get more idle jobs if jobs are evicted
  - **`Maxpre` *N*** limits PRE scripts
  - **`Maxpost` *N*** limits POST scripts
- All limits are *per DAGMan*, not global for the pool or submit machine (sub-DAGs count separately)
- Limits can be specified as arguments to **`condor_submit_dag`** or in configuration

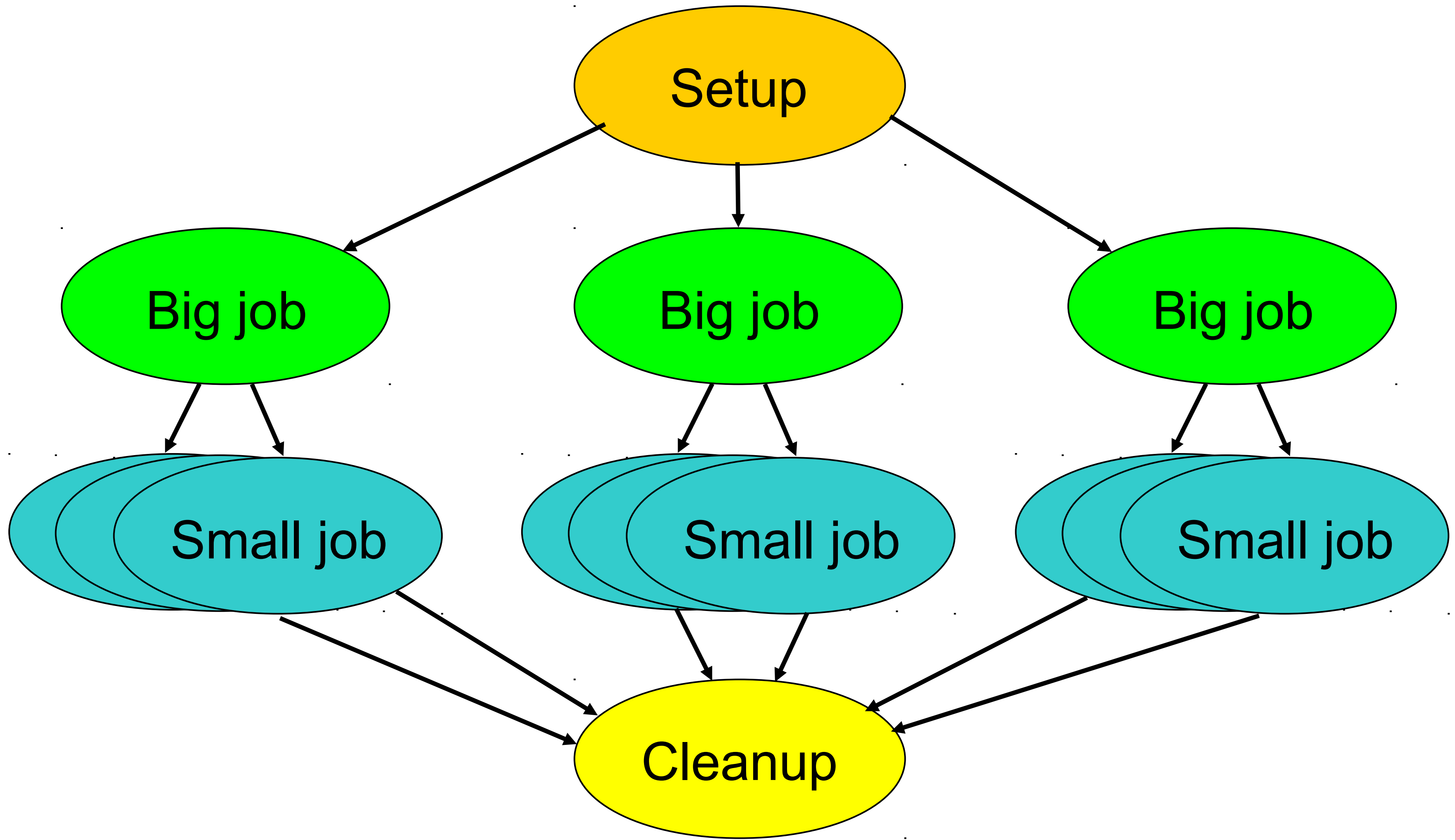
Throttling (cont)

- Example with per-DAG config file

```
# file name: foo.dag
CONFIG foo.config

# file name: foo.config
DAGMAN_MAX_JOBS_SUBMITTED = 100
DAGMAN_MAX_JOBS_IDLE = 5
DAGMAN_MAX_PRE_SCRIPTS = 3
DAGMAN_MAX_POST_SCRIPTS = 15
```

Finer-grained throttling



Node category throttles

- Useful with different types of jobs that cause different loads
- In the DAG input file:

```
CATEGORY JobName CategoryName
MAXJOBS CategoryName MaxJobsValue
```

- Applies the *MaxJobsValue* setting to only jobs assigned to the given category
- Global throttles still apply

Cross-splice node categories

- Prefix category name with "+"

```
MaxJobs +init 2
Category A +init
```

- Set **`MaxJobs`** in top-level DAG
- Assign nodes to categories within splices
- See the Splice section in the manual for details

Submit file re-use: node variables

- To re-use submit files for multiple nodes
- In DAG input file:
  VARS *JobName varname*="*value*" [*varname*="*value*"...]
- In submit description file:
  $(*varname*)
- ***varname*** can only contain alphanumeric characters and underscore
- ***varname*** cannot begin with "**queue**"
- ***varname*** is not case-sensitive
- ***varname*** beginning with "+" defines ClassAd attribute (e.g., **+State = "Wisconsin"**)

Node variables (cont)

- Double quotes in `Value` must be escaped
- The variable `$(JOB)` contains the DAG node name
- `$(RETRY)` contains retry count
- Any number of VARS values per node
- DAGMan warns (in `dagman.out`) if a VAR name is defined more than once for a node

Node variables (ex)

```
# foo.dag
Job B10 B.sub
Vars B10 infile="B_in.10"
Vars B10 +myattr="4321"

# B.sub
input = $(infile)
arguments = $$([myattr])
```

Tuning DAGMan: DAGMan configuration

- A few dozen DAGMan-specific configuration macros (see the manual…)
- From lowest to highest precedence
  - HTCondor configuration files
  - User's environment variables:
    - **`_CONDOR_`*`macroname`***
  - DAG-specific configuration file (preferable)
  - **`condor_submit_dag`** command line (recorded in **`dagman.out`** file)

Per-DAG configuration

- In DAG input file:
  `CONFIG` ***`ConfigFileName`***
  or command line:
  `condor_submit_dag -config` ***`ConfigFileName`*** **`...`**
- Generally prefer **`CONFIG`** in DAG file over **`condor_submit_dag -config`** or individual arguments
- Specifying more than one configuration file is an error.

Per-DAG configuration (cont)

- Configuration entries not related to DAGMan are ignored
- Syntax like any other HTCondor config file

```
# file name: bar.dag
CONFIG bar.config

# file name: bar.config
DAGMAN_ALWAYS_RUN_POST = True
DAGMAN_MAX_SUBMIT_ATTEMPTS = 2
```

Optimize your workflow: node priorities

- In the DAG input file:
  `PRIORITY` ***`JobName PriorityValue`***
- Determines order of submission of ready nodes
- DAG node priorities are propagated to job priorities (including sub-DAGs)
- Does *not* violate or change DAG semantics
- Higher numerical value equals "better" priority

Node priorities (cont)

- Better priority nodes are not *guaranteed* to run first!
- Effective node prio = max(explicit node prio, parents' effective prios, DAG prio)
- For sub-DAGs, pretend that the sub-DAG is spliced in
- Overrides priority in node job submit file
- *Not* relative to other users

Bailing out: DAG abort

- In DAG input file:

  ABORT-DAG-ON ***JobName AbortExitValue***
  [RETURN ***DagReturnValue*****]**
- If node value is ***AbortExitValue***, the entire DAG is aborted *immediately*, implying that queued node jobs are removed, and a rescue DAG is created.
- Can be used for conditionally skipping nodes (especially with sub-DAGs)

Cleaning up: FINAL nodes

- FINAL node *always* runs at end of DAG (even on failure)
- Use: garbage collect intermediate files
- Use **`FINAL`** in place of **`JOB`** in DAG file:

```
FINAL NodeName SubmitFile
```

- At most one FINAL node per DAG
- FINAL nodes cannot have parents or children (but can have PRE/POST scripts)
- Cannot have retries, category or priority

FINAL nodes (cont)

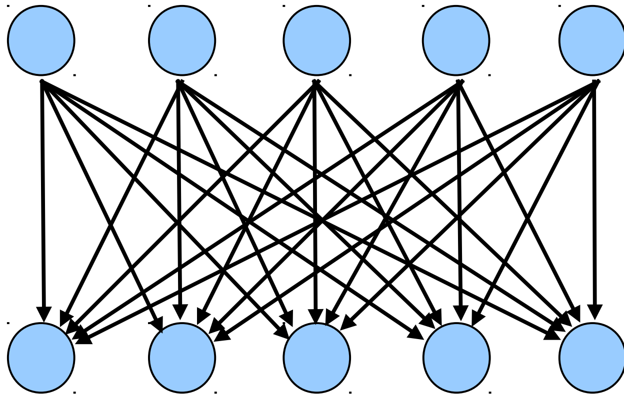
- Success or failure of the FINAL node determines the success of the entire DAG
- PRE and POST scripts of FINAL (and other) nodes can use **`$DAG_STATUS`** and **`$FAILED_COUNT`** to determine the state of the workflow
- **`$(DAG_STATUS)`** and **`$(FAILED_COUNT)`** available as VARS

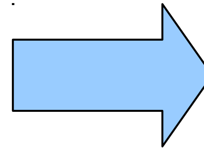
Doing nothing: no-op nodes

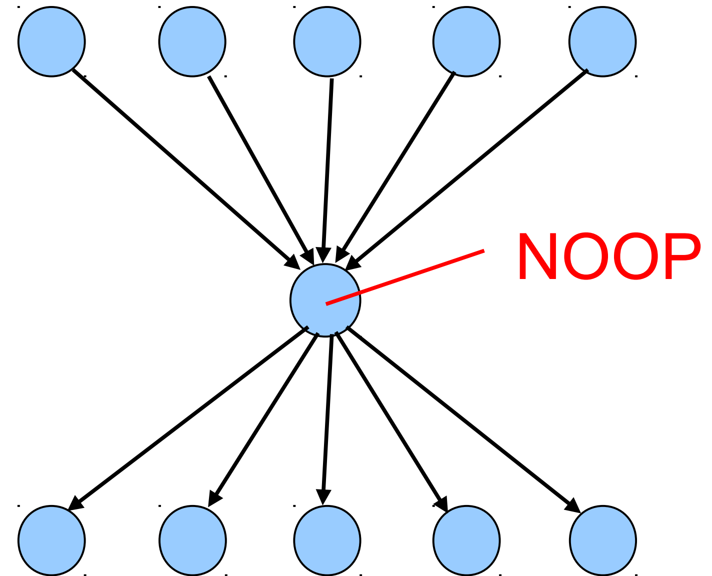
- Appending the keyword **NOOP** causes a job to not be run, without affecting the DAG structure.
- The PRE and POST scripts of NOOP nodes will be run. If this is not desired, comment them out.
- Can be used to test DAG structure
- Also avoid “combinatorial explosion” of dependencies

No-op nodes (ex)

Simplify DAG structure







No-op nodes (ex)

- Here is an example:

```
# file name: diamond.dag
Job A a.submit NOOP
Job B b.submit NOOP
Job C c.submit NOOP
Job D d.submit NOOP
Parent A Child B C
Parent B C Child D
```

- Submitting this to DAGMan will cause DAGMan to exercise the DAG, without actually running node jobs.

There's lots more...

- See the DAGMan chapter of the HTCondor manual: http://research.cs.wisc.edu/htcondor/manual/v8.5/2_10DAGMan_Applications.html
- Talk to me (Kent Wenger) some time this week
- For more questions: htcondor-admin@cs.wisc.edu, htcondor-users@cs.wisc.edu