# JEDI-alpha status and plans

Tadashi Maeno (BNL)

# JEDI

- Making the system more task-oriented
  - Tasks are submitted to the system
  - JEDI optimizes job parameters and generates jobs on behalf of users
  - Task-level scheduling
    - retry, rebrokerge, merging, …
- Beneficial to users
  - Users are interested in tasks rather than jobs
  - Simplify client tools and centralize user's functions
  - Can use computing resources optimally without detailed knowledge on the entire system
- JEDI-$\alpha$ = Alpha version of JEDI
  - Minimum (limited) functions
  - Focused on production
  - Co-exist with the current workflow
    - AKTR $\rightarrow$ converter $\rightarrow$ DEFT DB $\rightarrow$ JEDI-$\alpha$ $\rightarrow$ Panda
    - AKTR $\rightarrow$ prodDB $\rightarrow$ Bamboo $\rightarrow$ Panda
  - To have an early prototype with real workload for incremental development of JEDI

# Design Concepts for JEDI 1/3

- Structural Partitioning
  - Source
    - Production/analysis/test/…
  - VO
    - atlas/cms/ams/…
  - Function
    - Discussed later
- Logical partitioning in each structural partition
  - Production
    - Cloud
    - Work Queue
      https://twiki.cern.ch/twiki/bin/viewauth/Atlas/PandaJEDI#Work_queues
  - Analysis
    - User
- multiprocessing rather than multi-threading
  - $N_A$ worker processes for partition A, $N_B$ worker processes for partition B, …
  - Each worker process runs independently for a partition but processes share connections to DDM and DB
- Strict control over the number of connections to DDM and DB
  - Connection pools composed of multiple daemons
  - Worker processes communicate with connection daemons

# Design Concepts for JEDI 2/3

```
Line
   1 ##########################
   2 #
   3 # Database parameters
   4 #
   5
   6 [db]
   7
   8 # host
   9 dbhost = ADCR_PANDA
  10
  11 # user
  12 dbuser = ATLAS_PANDA_WRITER
  13
  14 # password
  15 dbpasswd = FIXME
  16
  17 # database
  18 dbname = PandaDB
  19
  20 # number of task buffer instances
  21 nWorkers = 5
  22
  23
  24
  25
  26 ##########################
  27 #
  28 # DDM parameters
  29 #
  30
  31 [ddm]
  32
  33 # interface config
  34 modConfig = atlas:3:pandajedi.jediddm.AtlasDDMClient:AtlasDDMClient
  35
  36 # list of VOs which use scope
  37 voWithScope = atlas
  38
```

- Configurability
  - panda_jedi.cfg
    - Written in the style of RFC822 to be readable using the standard python ConfigParser module

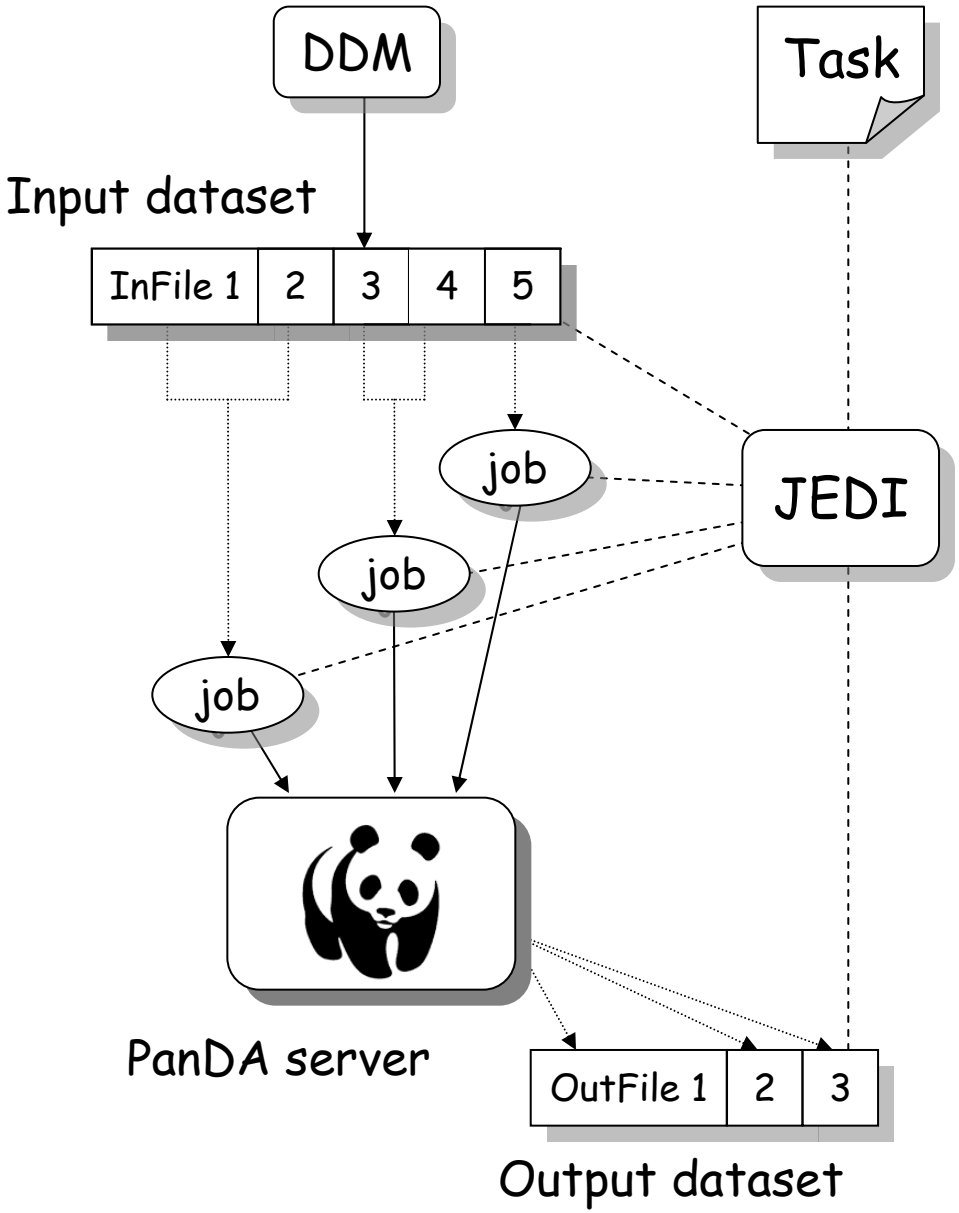
VO:nProcesses:ModueName:ClassName
Can define how each VO accesses their DDM system

- Easy to add new module/class without changing JEDI core

# Design Concepts for JEDI 3/3

- Plug-in architecture
  - Traditional interface + factory pattern
  - Factories instantiate concrete objects using panda_jedi.cfg where module and class names for the objects are specified
    - Module/class names are not hardcoded
  - Concrete objects and communication channels are instantiated in separate processes if necessary
    - Easy to have multi-processing structure
    - Worker processes share the interface
      - Invocation of an interface method is transparently converted to an IPC request and sent to a concrete object through a communication channel
    - The number of concrete objects is under control
  - The idea is to generalize the system to support multiple VOs

# Workflow of JEDI




1. Task is submitted to the system
2. Dataset contents are retrieved from DDM
3. JEDI registers output dataset
4. JEDI splits input and generates jobs
5. Jobs are submitted to panda server
6. Files are added to output dataset
7. Task is complete when all input files are processed
8. Output dataset is frozen

# Minimum Functions for JEDI-α

- Task Refiner
  - Gets and parses task parameters from DEFT to fill JEDI tables
- Contents Feeder
  - Retrieves dataset contents from DDM
- Job Generator
  - Throttles job submission if there are enough jobs in Panda
  - Selects site candidates using several matchmaking
  - Splits input
  - Generates jobs
  - Sends jobs to Panda
- Job Status Synchronizer
  - Updates JEDI tables when job status is changed in Panda
- Post Processor
  - Optimizes job parameters using scout job metrics
  - Makes job avalanche once scout jobs finish
  - Finishs tasks once all files are processed

# Parsing of Task Parameters 1/2

- DEFT gives a schema-less object to JEDI as task parameters
  - Written in JSON
- Why schema-less
  - Task parameters could heavily depend on task types
    - Production vs analysis
    - Special tasks. E.g. tasks for lost file recovery could give only a list of lost files as a task parameter
  - Flexibility and easy maintenance
    - Overkill to add new columns to DB every time new parameters are needed

# Parsing of Task Parameters 2/2

- JEDI selects an appropriate parser to fill JEDI tables based on VO, source, task type
- Example of task parameters

  https://twiki.cern.ch/twiki/bin/viewauth/Atlas/DeftJedi#Task_Parameters
- Extendability
  - Parsers are plugins in TaskRefiner
  - Just add a new plugin for a new usecase

# Interactions with DDM

- Decouple DDM interactions from the main body of JEDI
  - One connection pool with several daemons is spawned for each VO
  - Worker processes communicate with daemons using IPC
- Each daemon imports a DDM client module for a VO in own memory space
  - DDM modules don't conflict if JEDI works for multiple VOs
- For DQ2 → Rucio migration, just replace daemons

# Job Generation 1/2

- Brokerage is done at the start in the new workflow
  - Old
    - Splitting → Job Generation → Submission → Brokerage
  - New
    - Brokerage → Splitting → Job Generation → Submission
  - To optimize splitting to utilize CPU/memory/disk resources efficiently at the assigned site
  - This is already the case for analysis
    - Done on the client side now → will be done on the server side

# Job Generation 2/2

- Scout jobs give the following job metrics per file and per event for tasks with file-level and event-level splitting, respectively
  - Memory consumption (R)
  - Output size (O)
  - Scratch disk usage (W)
  - Execution time (T)
- Input is split to meet the following conditions
  - R < schedconfig.memory
  - $(W+O) \times N_{\text{file or event}}$ + input size < schedconfig.maxdir
  - $T \times N_{\text{file or event}}$ < schedconfig.walltime

# Current Status and Plans

# Current Status

- Minimum functions of JEDI-$\alpha$ listed in page 6 have been implemented
  - The whole task workflow with file-level splitting and event-level splitting
  - Special treatment for DBR
  - Support for secondary datasets (min-bias, cavern, …)
  - The scout $\rightarrow$ avalanche chain
  - Single master + multiple workers
- Scalability test of JEDI-$\alpha$ on INTR + 1 SLC6 vobox
  - Successfully scheduled 1.5 million jobs per day
- Dmitry Golubkov has developed an app to convert AKTR $\rightarrow$ DEFT.task_params

# Near Term Development Plans

- Improve splitting to be more intelligent
  - E.g., taking lumi block boundaries into account
- Implement task brokerage
  - Currently clouds for tasks have to be preassigned
- Analysis task
- Event Server
  - See Torre's talk
- Support for variable number of output files for AthenaMP
  - Next slide

# Variable Number of Output Files 1/3

- ATLAS A-team people have requested this functionality for AthenaMP-2
  - AthenaMP-2 has a flexibility to skip the merging step → one job produces multiple output files
- Two options
  - Give a list of output filenames to AthenaMP before the job gets started
    - The number of filenames would be the same as the number of cores
    - Essentially no deference from current reco jobs which produces ESD,AOD,TAG,NTUPs…
    - Some changes are required in AthenaMP and TRF
  - Give a pattern to AthenaMP to produce files accordingly, and then the system regards files matting the pattern as outputs
    - Preferable to A-team

# Variable Number of Output Files 2/3

- Essentially job definition is changed when job finishes
- There is the same request for analysis
  - Currently the --output option allows wildcards, e.g., "--output blah.root.*"
  - Files (blah.root.*) are save into a single archive (tar.tgz) and the archive is added to DQ2
  - Some people want to have root files instead of tgz files since the former is convenient for subsequent jobs

# Variable Number of Output Files 3/3

- Required changes to the system
  - Introduce "outpattern" to JEDI as an output type
  - Change the panda sever to be aware of outpattern
  - outpattern is implanted in jobPrams
    - E.g, outputHitsFile=HITS.XYZ._001.pool.root → outputHitsFile=HITS.XYZ._001.*.pool.root
  - AthenaMP or trf produces a json which contains the list of output files mathing the pattern
  - The pilot extract file info from the json to add them to output XML and sends it to the panda server
  - The panda server inserts file records to pandaDB and registers them to DDM
  - The file info is propagated to JEDI

# Migration Plan 1/2

1. Change the panda server for job.taskID
   - Cannot make FK between job tables and JEDI_Tasks table for taskID
   - Could introduce a new column to job tables
2. Add several changes to existing Panda database tables
   - See Gancho's talk
3. Add JEDI tables and define workQueues in JEDI_WorkQueue
4. Change Bamboo to fill job.workQueue_ID
   - The number of queued or running jobs can be calculated per workQueue → Throttling can work for each workQueue

# Migration Plan 2/2

5. Set job.lockedBy=jedi in JEDI
   - Bamboo ignores JEDI jobs since it sees jobs with lockedBy=bamboo
6. Change the Panda server to use new columns added in step.2 when job.lockedBy=jedi
7. Change schedconfig.fairsharepolicy from type=??,group=??:?? to type=??,type=??,id=??:??
   - Panda brokerage or dispatcher ignores "id=???" while JEDI uses only "id=???"
8. JEDI can send jobs

# Test Plan

- Using real tasks
- Will be discussed this afternoon