SuperTask WG Input - Batch Processing Service High Level

The following is an attempt to describe the Batch Processing Service at a level to inspire SuperTask working group discussions. Many internal details of batch processing have been omitted from this document in order to focus on items relevant to the Supertask WG discussions. Also we tried to minimize explicit assigning of statements with the exception of science pipelines must use a Butler repository. After an initial walk through of this page in a WG meeting, it is expected that several pieces will need to be detailed further.

Things to keep in mind while reading

- L1 alert prompt processing is allowed to have its own special system in order to meet time requirements. (Unclear whether L1 offline would use normal Batch Processing Service)
- Batch Processing Service needs to be able to run jobs on generic HPC clusters that will not direct access to a pre-existing Butler repository (as one could if running on their workstation or against /datasets on the lsst-dev cluster). Files will need to be staged.
- Strong preference to be able to run pipelines without direct access to central DB
    - Some hpc clusters will not allow outgoing connections
    - Easier for development and tests if doesn't need central DB
    - Helps limit simultaneous connections to central DB especially in L2 processing case
    - More difficult to automatically restart a process if it has already modified DB tables
- Operations DB is different than the Release DB
    - Different schemas optimized for different use cases - e.g., read & write vs write once & read many
    - Operations DB is allowed to have data from test runs, failed runs, etc so queries need to be more selective.
- Requiring unique filenames (at minimum by the time Data Backbone saves file, metadata, and provenance)

Batch Processing

1. Operator configures campaign (one or more executions of pipelines)
    a. Sets values needed to determine calibration sets, input sets, etc
    b. Sets "sequence" of steps (TBD: determine pipeline language for operator)
2. Operator submits campaign configuration (TBD: including the butler mapper policy in order to name output files (how flexible is butler file naming i. e., can new variables/data ids be added without code changes) and more)
3. Campaign manager/Operator enqueues pipelines for execution

    (let's examine a single pipeline)

4. Save submission/framework config file(s) as provenance for pipeline (possibly just as data files)
5. Converts generic/abstract pipeline description to expanded/specific pipeline description/graph that contains exact number of steps, dataset types, data ids and filenames
    a. If cannot determine inputs for a step until after a previous step has finished, this is a separate sub-graph of the pipeline that starts back over at step 5
6. Using expanded pipeline graph, get input filenames and call DBB to pre-stage input files from tape to disk within Data Backbone.
7. If need to convert DB data into files, do so now (e.g., reference catalog)
    a. Note: May be sqlite files to still allow DB-style operations
    b. These files should be tracked in DBB for provenance.
8. If shared filesystem available at computing site, pre-stage input files to shared filesystem.
    a. Any file tracking system for the computing site cache needs to be updated
9. Create workflow (needs input filenames and unique output filenames for each step)
10. Allocate compute resources
11. Within job on compute node
    a. Stage input data (file and metadata file) to compute node (could be to local disk)
        i. If shared filesystem available, create symlinks to files or copy files to local disk (which ever works better for the compute site)
        ii. If no shared filesystem available, will need to retrieve from central Data Backbone
    b. Gather job provenance (information such as actual job environment that cannot be done outside of a job)
    c. Initialize job-specific input butler repo (for each input file, needs mapping of data id + dataset type to where currently located. In some cases also need "metadata" (TBD) to initialize the input butler)
    d. Runs task sending output data to output butler repo
    e. Put files in workflow friendly directory structure if workflow isn't going to manage files directly in output butler repo
    f. Gather information about each output file (including log files):
        i. (should already know data id + dataset type)
        ii. science metadata (based upon full dataset type (must uniquely determine what metadata values to gather – if dataset type is not enough to uniquely determine this, we need another "type" label))
        iii. physical attributes (filesize, md5sum)
        iv. provenance (minimum, was-generated-by which ties files to the step that generated them)
        v. (optional – operator setting) Create junk tarball of output files not known to be valid output file
    g. Gather runtime stats for each step (e.g., start/end times, memory usage, etc)
    h. Stage output data out from compute node
        i. If shared filesystem available, local copy operation
            1. If shared filesystem is tracked by data backbone, DBB's file catalog needs to be updated (TBD – file catalog centralized (e.g., oracle DB), local to shared filesystem (e.g., sqlite3), or some combination of both.
        ii. If no shared filesystem available, will need to push files to central Data Backbone.
12. End-pipeline: Ingest object catalogs, logs, metadata, file provenance, etc into appropriate DB/services if not done from within job.
13. Repeat from 5 if needed (for portions of pipeline that need cannot determine work until after previous steps complete)
14. End-pipeline: Stage output files from shared filesystem back to central DBB.
15. Clean up shared filesystem post-pipeline (intermediate files within pipeline)
16. Clean up shared filesystem end campaign (files shared by pipelines)

Example Operations Needs (Not complete list)

- Choosing what steps will be done in a compute job and where that compute job will run
- Need to understand resources needed to run step (e.g., memory, cores)
- Need ability to set level of parallelism to be done inside a step.
- Need ability to turn off unwanted features in Butler (and Supertask)
    - Example: No file transfers from Data Backbone within job
- Need to be able to detect failures
    - Need ability to group like failures together across pipeline executions
- Ability to track all inputs and outputs of a particular pipeline execution
    - Find all (non-junk) executions that used a particular input file
    - Ignore/Find all entries in DB table X that were inserted as a result of a particular pipeline execution.
- Ability to find patterns in failures or irregular behaviors (e.g., running much slower)
    - particular computing site
    - particular computing node (e.g., need to avoid node0003 for current submissions and make report to admins)
    - particular time frame
    - particular science information (e.g., same band)
    - particular step in pipeline (the more fine grained the information should mean easier debugging)
    - (Basically joining any table to any table in consolidated DB given some starting information.)
    - (Currently assuming after the fact as no outgoing information from inside a job.)
- Ability to change configuration of pipelines
    - Normal science configuration values
    - Add steps
    - Remove steps (e.g., stop pipeline after step X, remove step Y from pipeline)
    - Reorder steps
- Ability to change/filter inputs
    - Avoid bad or test files (e.g., using tags or blacklist)
    - For a particular kind of test, use a subset of inputs to speed up test.
    - Test with an “unusual” input (e.g., different calibration file)
- Ability to turn on/off saving of intermediate files
    - Across the board
    - For a particular step
- Easily configure logging for pipeline
    - Name each log file (want separate logs for each step)
- Ability to send parts of pipeline to one site and later parts to a different site
- Ability to recall how a particular step of a pipeline was executed (configuration, inputs, etc).
- Ability to rerun pipeline as it was previously run:
    - Same code (including framework and services) and inputs
    - Or perhaps science code the same, but services and framework can be current versions.
- Ability to use (efficiently) various computing platforms
    - On systems with externally visible file systems, will want to stage files without using a remote compute job to do so
    -

Notes:

- When choosing when a particular process is going to be done (e.g., gathering job level environment), we should keep in mind what is the optimal place to do so
    - How often does the process really need to be done especially in the context of a processing campaign?
    - As a general rule processes/software should not monitor themselves
    - If a process is to gather information, ensure that we get enough information when there are problems (i.e., call the gather information early enough)