Gen3 Middleware Camera Specialization Interfaces

*This page frequently refers to **DMTN-073**, which is currently in progress on*

⚠ *DM-12620 - Jira project doesn't exist or you don't have permission to view it.*

. Actions that involve reviewing its content should be considered blocked by that ticket.

Overview of Relevant Changes

Data Models for Observational Data

In the Gen2 Butler, we permit each camera to have its own data model for observational data - it can have its own system for labeling exposures, sensors, etc. This makes things very natural at the very highest level when we want to specify which units of data should be processed or analyzed, but it is a problem for pipeline developers in two ways:

- There is no camera-generic way to express a concept like "visit", even though grouping by such concepts is frequently necessary in pipeline code.
- Each camera must provide its own path template for any dataset that includes data ID keys related to observational data, making the definition of new DatasetTypes much heavier.

In the Gen3 Butler, we instead expect each camera to conform itself to a common data model: Exposures, Visits, and Sensors are defined as generic concepts that pipeline code can utilize, and cameras must provide their own definitions of these terms that are consistent with the generic concept.

Data IDs DataUnits

The flexible dictionary data IDs used in the Gen2 Butler have been both generalized and more strictly enumerated into what we call *DataUnits* (for "Units of Data") in the Gen3 Butler. Dictionary data IDs will still be used, but the keys allowed in those dictionaries are limited to a predefined set of DataUnit types, and we are making a much stronger distinction between keys that uniquely identify certain DataUnits (such as the unique ID number associated with a visit or tract or the name of a filter) with metadata that can be used to look them up (such as the date a particular visit was observed). The latter can no longer be used in data IDs, as we are making a strong separation between "complete" data IDs that can uniquely identify a dataset and *expressions* that in general yield multiple datasets (they may also yield a single dataset, but this cannot be easily guaranteed before the query is actually performed). The Gen2 concept of a "partial" data ID no longer exists. In exchange, expressions will be much more powerful: they will support at least a significant subset of SQL.

A full description of the set of DataUnits is beyond the scope of this document (see DMTN-073). The relevant ones for this document will be discussed at various points below. Some other important facts about DataUnits:

- DataUnits can have dependencies: for example, a Patch is always associated with a particular Tract.
- Each DataUnit has one or more *value fields*, which uniquely identify it *when combined with the value fields of the DataUnits they depend on*. For example, a Patch's (x,y) index uniquely identifies it only when combined with its Tract's number (and, now, its SkyMap's name, since SkyMaps are themselves DataUnits). A data ID like `(skymap="rings-120", tract=8766, patch=(4,5))` thus uniquely identifies a SkyMap DataUnit, a Tract DataUnit, and a Patch DataUnit.
- *Some* DataUnits are associated with a table in the Registry that provides additional metadata (such as the timestamp or airmass of a Visit) as well as relationships between DataUnits (such as the PhysicalFilter associated with a Visit).
- We often need to be able to assign a unique *reversible* integer ID to a particular combination of DataUnits. For example, when generating ObjectIDs, we want to combine an auto-increment number within a particular tract and patch with a unique identifier for that tract-patch combination. That means we need a way to generate those IDs from many different combinations of DataUnits, and a natural way to do this is for each DataUnit to have a mapping to an integer and a way to get at the maximum number of bytes or bits that integer occupies (much like we do in Gen2 with e.g. `bypass_ccdExposureId`).

No Mappers, No Special Mapping Hooks

There will be no mappers in the Gen3 Butler. The template used to write a dataset (assuming files are even involved) with a particular DatasetType will usually be defined by the Butler client configuration (though this could load default configuration from a repository). Camera-level template specializations for outputs may be supported, but we don't expect them to be used very often. The concept of a template for reading will no longer be meaningful - we will store the actual filename (or equivalent) of every dataset we store in a database, and look it up directly (as a result, there are no raw data templates, and hence no need to camera-specialize those).

The special mapping hooks (`std_`, `bypass_`, etc) that are used in the Gen2 Butler to override how certain datasets are read will be gone as well. Instead, composite datasets and a more flexible system for specifying Formatter objects that do the writing (and automatically saving the Formatter for reading) will be used for many of these customizations. In Gen3, these customizations are generally applied when datasets are ingested or otherwise written, and they are stored with the datasets (at the level of individual datasets, that is; this metadata is not necessarily on the same e.g. disk as the dataset). That means no special configuration should be needed in a Butler client to read something once it has been persisted, but it also means that it is much more difficult to change how a dataset is read after it has been written. That limitation suggests a model in which raw data is *not* automatically augmented with auxiliary information on read, but is instead augmented by the creation of "virtual" (no-new-files) datasets that contain the auxiliaries during (or even prior to) ISR. This may in some contexts be less convenient than what we have now, but I believe it is the only way to rigorously capture the provenance of that auxiliary information (e.g. which version of what the camera geometry looked like at at a particular point in time should be associated with a raw dataset).

Data Repositories vs. Collections

In the Gen2 Butler, different processing runs are represented by different data repositories that were mapped to different directories, and the relationship between inputs and outputs are represented by chained repositories that are searched in a particular order.

In Gen3, a single data repository will contain both raw data (possibly from multiple cameras) and a large number of processing runs. Different processing runs or manually-curated groups of related datasets are represented by Collections, which are simply a database tag (they have *no* representation in filesystem/Datastore). Datasets can belong to any number of Collections, and typically the Collection associated with a processing Run will include all Datasets in the Collection(s) that were used as an input to that processing Run. With a Collection, however, there can only be a single Dataset with a particular DatasetType and Data ID. A Butler instance is associated with only one Collection, so any `Butler.get` call should have a unique and unambiguous result.

Populating Instrumental DataUnits

Gen3 data repositories will not be limited to data from a single Camera, and this means that Camera is itself a DataUnit; a data ID that includes observational data must include a key-value pair that identifies the camera (by a short string name, like "HSC" or "DECam").[1]

Each Camera is also responsible for defining sets of associated Sensor and PhysicalFilter DataUnits.

- [ ] **For Obs WG:** review the proposed schemas for Camera, Sensor, and PhysicalFilter in DMTN-073. Note that the Gen3 design also permits *additi onal* per-Camera metadata tables for DataUnits, but we'd prefer to put as much as possible in the common, camera-generic DataUnit tables (and ideally avoid having any per-Camera metadata for as many DataUnits as possible).
- [ ] **For Obs WG / Gen3 MW Team:** design a class interface that obs_* packages can specialize to provide descriptions of Cameras, Sensors, and PhysicalFilters. This should include:
    - What needs to be provided when adding a Camera to a Registry (i.e. adding entries for that Camera to the Sensor and PhysicalFilter tables for the first time).
    - Keeping the Registry information up-to-date (e.g. what happens when a new filter is added).
    - How to convert Sensor and PhysicalFilter DataUnits to unique integer IDs and back, along with a description of how many bits/bytes those integers occupy for this Camera.

1. We'll try to to infer the camera name whenever possible, and make it usually unnecessary to include it in high-level, user-facing code. But pipeline code will typically work with data IDs that do include the camera explicitly (though those IDs will also be considered opaque, so they shouldn't care).

Producing and Registering Master Calibration Datasets

Pipeline-Produced Master Calibrations

The Gen3 does not grant any special status to calibration repositories. The *set* of master calibrations used in a processing run is selected by including the the Collection containing the desired calibrations as one of the input Collections for the Run, which is no different from how, say, a Collection of processed visit images is used as inputs to a coaddition processing run.[2] The actual master calibration datasets used for a particular (e.g.) science frame is identified by the relationships between the calibration dataset's DataUnits and those of the image being calibrated. Instrumental DataUnits - Camera, PhysicalFilter, Sensor - must match exactly (though not all need to be present; some calibrations are not associated with a particular Sensor or PhysicalFilter, and some are associated with neither). In addition to these, most master calibration products are also labeled with an ExposureRange DataUnit, a compound unit that is defined by a pair of `valid_first` and `valid_last` Exposure IDs. These are naturally related to the Exposure DataUnit that labels (e.g.) a raw science image; any calibration data products that have an ExposureRange that includes the science data product's Exposure are automatically associated with it, in essentially the same way that spatially overlapping Visits and Tracts are associated (both are intrinsic, range-based, many-to-many relations; the only difference is whether the range is temporal or spatial). This has three implications:

- Cameras must define Exposure IDs to be monotonically increasing with time (this is obviously highly desirable anyway).
- It is the responsibility of the calibration products production and registration system to ensure that when a calibration product lookup for a particular raw science image must be unique, it is (within a Collection). Because it is possible that we will have some calibration products for which it may make sense to associate multiple such products with a single raw science data product (within a single calibration Collection), the Butler itself makes no such restriction.
- Master calibration products have their own data IDs (involving `valid_first` and `valid_last` keys, typically), and cannot be retreived by calling `Butler.get` with the data ID of a raw science frame they are associated with. This should be completely transparent to anyone implementing a SuperTask that uses calibration products (e.g. ISR) - their `runQuantum` methods will be automatically provided with an opaque handle object for each input dataset they use, grouped by DatasetType, and they'll just use those to get what they need. Analysis code that needs to look up a calibration dataset from a raw science data ID would have to go through a two-step process that could in general yield multiple results, but it's worth noting that this should be much more rare than it is now, because looking up which calibration dataset *was actually used* to generate a particular output will be a query we can support through the provenance system.

Pipelines that *create* master calibration products frequently use the reverse of this relation between Exposures and ExposureRanges: they take as inputs raw datasets identfied with Exposure DataUnits (e.g. raw flat observations) and produce, as outputs, datasets identified with ExposureRange DataUnits (as well as various instrumental DataUnits, of course). It will also be possible to provide a custom mapping from input Exposures to output ExposureRanges by configuring a SuperTask with custom SQL (details still TBD).

It is worth noting that calibration product generation is a major use case for supporting data from multiple Cameras in the same data repository and Pipeline: associated instruments like LSST's auxiliary telescope and Collimated Beam Projector[3] will be represented as different Cameras, and the raw data from with them will naturally be combined with raw data from the main LSST Camera to produce master calibration datasets for the main LSST camera.

- [ ] **For Obs WG / CPP Team:** review the Exposure/ExposureRange Data Unit joins described in DMTN-073 to make sure they can express the relationships between master calibration products and both raw science frames and raw calibration frames.

The additional columns in the Exposure DataUnit table are expected to provide everything we might want to filter on when specifying the inputs to calibration products production pipelines on the command-line. That cannot and should not be everything in the EFD, but copying information from the EFD to the Exposure DataUnit table is not the only way to make it available via the Butler. We can also imagine bundling EFD data into actual butler-managed datasets, as long as we can use the DataUnit system to label them. Those could be used to further filter inputs (as can the properties of the input datasets themselves) within the Task code.

- [ ] **For Obs WG / CPP Team:** review the schema of the Exposure DataUnit table in DMTN-073 to make sure it contains everything we'd use to select inputs to calibration product pipelines. The schema will not be set in stone, of course, but changes will become increasingly disruptive as Gen3 Butler usage grows. As with instrumental DataUnits, we can also have per-Camera columns for Exposure metadata, but we prefer to put as much in the common table as possible.

2. We will again provide high-level convenience code (in e.g. the driver code that initiates the execution of SuperTask Pipelines) to make sure the appropriate calibration Collection is used when none is provided, but there will be no distinction between Collections that contain calibration datasets and Calibrations that contain other kinds of datasets (or Collections that contain both) at the Butler level.

3. The CBP is of course not actually a camera, but because it's essentially the inverse of one it still maps moderately nicely to this data model.

Human-Curated Master Calibrations

Some master calibration products are not produced by a pipeline, and are instead essentially human-curated. This includes defect lists, transmission curves (as they are implemented today), and camera geometry (in the future). The representation of these datatasets in a Gen3 Butler repository should be no different from pipeline-produced calibration datasets: they should be identifed by ExposureRanges and instrumental DataUnits, and included in Collections containing other calibration datasets in a way that ensures unique lookups for raw science exposures when appropriate.

Datasets managed by the Gen3 Butler should never be modified in-place, however, as this makes rigorous provenance impossible, and this means that these datasets should be *created* from another representation that is considered the source of truth for the curation process. Git repositories seem like a natural approach, but I think that's entirely up to the Obs WG.

- [ ] **For Obs WG / CPP Team:** design a system for maintaining human-curated calibration datasets that is independent of the Gen3 Butler, and provide scripts or other tools for writing them to Gen3 Butler data repositories (possibly with some format/organization transformation).

Raw Data Ingest and Observational DataUnits

The Gen3 Butler defines two DataUnits that represent observations: Exposures and Visits.

- Exposures represent single observations with a camera. Exposures can correspond to raw science images or raw calibration frames. All datasets that represent ingested observations (rather than something produced from them) should be labeled with an Exposure DataUnit (they may of course be labeled with another DataUnit, such as a Sensor). Exposures depend on a particular Camera and may be associated with a PhysicalFilter.
- Visits represent a set of consecutive science Exposures that share the same pointing and PhysicalFilter. Visits are used instead of Exposures for higher-level datasets, even for data taken with an observational strategy that yields exactly one Visit for every Exposure, to enable the same code to be used for multi-Exposure Visits without change. Visits also depend on a particular Camera and are always associated with a PhysicalFilter.

Because both of these depend on Cameras (i.e. any Data ID with an `exposure` or `visit` key must also have a `camera` key), the numbering systems used to identify them are only unique within a Camera. The same Camera DataUnit specialization interface that defines how to translate a Sensor or PhysicalFilter into a unique integer must also do the same for Visits and Exposures. Because both Visits and Exposures are already identified by unique integers in the DataUnit schema, this "translation" is usually a no-op, but Cameras must also provide the number of bits/bytes those integers may occupy, which can be a bit trickier.

- [ ] **For Obs WG:** review the proposed schemas for Visit and Exposure in DMTN-073 and ensure that these have the necessary columns to express any query we might want to execute to define the inputs and/or outputs of a processing run. As usual, we want to encourage camera-generic columns whenever possible.
- [ ] **For Obs WG:** provide interfaces for translating Exposures and Visits to/from integer IDs with a Camera-specific maximum number of bits/bytes.

Raw ingest to Gen3 repositories is not appreciably different from raw ingest into Gen2 repositories. Ingest is responsible for:

- Adding entries to the Exposure and (if appropriate) Visit tables via a TBD Registry API.
- Adding entries to the Dataset table via a TBD Registry API.
- Actually transferring the file(s) to the appropriate location (if necessary) and notifying the Datastore of its existence via a TBD Datastore API.
- Adding entries to any Camera-specific Exposure and Visit metadata tables.
- Adding entires to any Dataset-specific metadata tables for the raw data.

How many of these operations are bundled into a smaller number of public middle API calls vs. called directly by ingest scripts is still TBD.

- [ ] **For Obs WG / Gen3 MW Team:** define interfaces for raw data ingest.

One major difference is that the Gen3 Registry also stores and indexes a spatial region for each Visit *and* each Visit+Sensor combination. These are expected to always be large enough to fully cover the observed area, but are not expected to represent it exactly; when using this information to relate the Visit or Visit+Sensor combination to Tracts and Patches, we always assume that some further refinement of those relationships will take place based on more accurate information. When possible, these regions should be populated during ingest, but when the known pointing is entirely unreliable, they may be left null. Regardless of whether they are provided at ingest, we expect to have later scripts that use pipeline outputs to update them (though they should only need to be updated once, given the accuracy required). For actual LSST data, for instance, it probably makes sense to set these after the initial AP processing of that Visit is complete, and then leave them unchanged.

Configuration/Pipeline Overrides

Because SuperTask Pipelines can in general process data from multiple cameras, the Gen2 approach of automatically applying Camera-specific configuration overrides doesn't really make sense. The (vague) approach I have in mind for Gen3 is for each obs_* package to also define one or more Pipelines, ideally by expressing the diff against some fiducial definition of a Pipeline. This could involve more than just configuration changes; it could also involve adding or removing entire SuperTasks. These Camera-specific Pipeline definitions would be passed directly or indirectly to the SuperTask execution driver code (where additional command-line overrides could be applied).

A Camera-specific Pipeline would still, of course, only be appropriate for data from that Camera, and I expect us to continue to predominantly process data from only one Camera at a time. This approach leaves room, however, for multi-Camera Pipelines to be defined in other packages (though these may be limited to just those steps that can be accomplished with the same configuration for all Cameras). It's also worth noting that this approach doesn't provide a way to stop users from accidentally using using a Pipeline configured for one Camera with data from another.

- [ ] **For Obs WG:** think about how we'd want to express a Camera-specific Pipeline as a diff against a fiducial Pipeline, and what (if anything) we can do to prevent Camera-specific Pipelines from being misused. Are there other approaches we could follow for providing customization of Pipelines for different Cameras?