Requirements and Design for Composite Datasets in Butler

Abstract

There need not be a one-to-one relationship between an instance of serialized data, such as in a single FITS file, and datasets (where the definition of a dataset is “the persisted form of an in-memory object. It could be a single item, a composite, or a collection”). The relationship can occur as many datasets in one file, and can also occur as one dataset spread across many files.

Many datasets in one file

A file (or other persistent representation of a dataset) may be composed of multiple component datasets. Each of those datasets should be retrievable independently, provided an appropriate plugin exists to read it.

This kind of functionality is currently provided for things like calexp_md (which reads just the headers from a FITS file). The Butler should continue to allow this. We don’t know of any particular additional features are needed here (Although per Jim Bosch, the procedure for defining new component datasets in mappers is pretty clunky, and is poorly documented.) Perhaps this could receive some attention as part of another story/epic if there is enough need.

One dataset across multiple files

A dataset being retrieved may be composed of multiple component datasets, each of which is its own independent dataset (and which could be a component of a Type 1 persisted dataset).

We need to add Butler support for getting python objects that are composed of datasets that exist in multiple files. This is the topic of this document.

Requirements

Composable and Decomposable

Objects that should be created from or, may contain, individual component objects must be able to be created from those separate components or have those components assigned at a later time. Those components may be persisted in the same dataset (e.g. FITS file) or different datasets.

Objects that should be persisted into individual components must provide a means of accessing those component objects for serialization. Right now there are no objects that are or can be composed of multiple datasets. Someone will have to write example serializers and deserializers.

Jim Bosch would like to jointly design the interface for these de/serializers with Nate, and then he thinks it is appropriate for Science Pipelines teams to write them for the most part. This idea needs to be discussed with John Swinbank and Simon Krughoff.

Pluggable

For components: Butler needs a mechanism for adding plugins that read and write component datasets.

For composites: Butler needs a mechanism for adding plugins that:

- Read and combine component datasets into an individual composite object.
- Write composite objects to separate component persisted datasets.
- Write composite objects to a single persisted dataset.

Distributed

Component datasets may exist within the same or different repositories.

Composite objects may or may not be persisted as single datasets. Writing a composite object to component datasets should be the default behavior where possible.

Implementation of this feature may not be implemented in the first release if the Composite Object can be serialized as a whole into a single file.

Policy In Repo

Datasets need to be able to change their definition (e.g. change the template) and new datasets need to be definable in repositories.

Datasets Findable by Processing Stage

Datasets must be findable according to what pipeline processing stage has processed the data.

- The Butler API should allow a processing level specifier for all datasets.
- Composite dataset definitions should be able to specify a level of processing for their component datasets.

API Proposals

The proposed solution in the use case example is to call the dataset type `calib:jointcal` (uses a colon separator).

I think we're going to have to require a limitation where if components to be used are from a different stage of processing (implicit in "different stage" is "stored in a different repository"), that the component search parameters (dataset type with optional processing level indicator and dataId) are stored in the registry where the composite (with processing state indicator) is found.

Component Access

The Butler API should support syntax to allow a component dataset to be retrieved by referencing a composite dataset without loading the composite dataset.

API Proposal

This relies on the proposal for Composite Dataset Type Definition (below).

The example is: given parameters to get a calexp, the code should be able to indicate that the returned object would be the component PSF of the calexp. The proposed API is to use dot notation to indicate the component type. I think this will work if the first item is the composite's <dataset type name> and the second item is the <component name> as indicated in the Composite Dataset Type Definition. That way, the api to get the Psf of the calexp would be b `utler.get('calexp.psf', ...)`.

I can image how this could use the policy to find the Exposure's getter to fetch the Psf out of a loaded Exposure, but I don't yet see how we can load the Psf without going through the mechanics of loading the entire Exposure. Its possible that butler code could figure out which Psf *would have* been loaded by the Exposure (but I haven't proved that to myself yet).

Additional DataId Keys in Later Processing Stages

Processed & stored datasets need to be able to have dataId keys that do not exist in precursor datasets.

Associated Object DataIds Persisted With Dataset

Objects need to be able to persist dataIds of associated components along with their object data.

Component DataId Extraction From Dataset Metadata

Plugins that read components to assemble composites need to be able to get dataIds from components, to find other component objects should be loaded.

Caching

Butler should cache Objects (instantiated objects, not datasets) so that they may be shared. Shared objects may be used as individual objects as well as components in a composite object.

Question: is the focus of caching to save memory, or to save read/write time? The (possibly) obvious answer is “both” but the point is to ask if the cache keeps a weakref to accessed component objects, will the refcount for objects that will be needed in the future generally be kept greater than zero by other owners that are not the cache?

Caching scheme will depend on requirements, TBD. Options include:

- weak_ptr/weakref is an option
  - needs prototyping for proof of concept across the SWIG boundary, represented by DM-7046
  - object is kept in cache only if another (non-butler) object holds a reference to the shared object.
- LRU Cache
- Clairvoyance - if the script can tell butler what dataset type + dataId will be used in advance, Butler may be able to determine in advance what objects should be cached.

Problem with Mutable objects

Blindly sharing objects via a cache is problematic if the objects are mutable. There is an ongoing discussion in this CLO thread. This point needs to be resolved.

C++ to Python Calls

C++ objects may need to be able to call back out to python to do the serialization, depending on factors such as if recursive serialization needs to call back out from C++ layer to Python for serializer registry lookup for component objects.

(How-to-implement note: C++ can call python by SWIG “directors”, or custom Python C API bindings can be written)

Other Related Issues

- When we add Dynamic Dataset Type creation, in the case of composite objects that are serialized as components, it will not be possible to write the definition of the Dynamic Dataset Type beside the dataset (the dataset will not exist in a single location). We will have to find another location for the persisted definition.

Questions & Issues

Existing Serializers & Deserializers

WTD re. Existing serialization schemes & plugins… ultimately we will need to refactor the existing serialization plugin interfaces to do composites well. Probably Jim & Nate (anyone else interested?) should discuss this more when we get to the design phase.

shared_ptr, weak_ptr, and weakref

Can we write C++ constructors and/or setters for composite objects that take swigged-and-unswigged shared_ptr to other C++ component objects? (I *think* so, but would like to be sure)

Implementation Idea(s)

This section contains notes & ideas (many or all of which relate to each other) about how we can implement these requirements. API requirements should not be expressed here (requirements for API should be documented in the Requirements section), but this section may be of interest to users in that how the system is implemented will have some effect on how it can be used, and may impose limitations that had not been considered in Requirements.

Composite Dataset Types

- Type 1: All the data for a single dataset is all the data in a single (FITS) file.
- Type 2: Many different datasets have data in a single (FITS) file.
- Type 3: A single dataset has data in many different (FITS) files.

Composite Dataset Type Definition

To indicate that a dataset should be serialized/deserialized from components, the policy's dataset definition has a new keyword `composite`.

```
The structure is:
<dataset type name>: {
    composite: {
        <component name>: {
            datasetType: <dataset type>
            setter: <method name of setter>
            getter: <method name of getter>
            assembler: <importable function to do custom deserialization>
            disassembler: <importable function to do custom serialization>
        }
        ...

    }
}
```

where:

- <dataset type name>: the name of the dataset
- composite: a new section. It is optional. (it should be omitted if the dataset is not a composite.)
- <component name> is a name that is used to refer to the component within the composite. Some default values can be inferred from the name (see setter & getter).
- datasetType: names the dataset type that should be used for this component.
- setter: names the method that is used to set the component in the composite class. Defaults to `set<component name>(component_object)`
- getter: similar to 'setter': names the method that is used to get the component from the composite class. Defaults to `get<component name>(component_object)`
- assembler: name of a function that can be used to instantiate the custom object if the default assembler (see the Assembler section below) is not suitable
- disassembler: similar to assembler but for custom deserialization.

Below is an example policy for the postISRCCD dataset from obs_cfht's MegacamMapper.paf, extended so that it is a composite where the WCS can be loaded separately. (I chose postISRCCD from obs_cfht's MegacamMapper.paf, only because it uses an ExposureF)

```
exposures : {
    postISRCCD: {
        template: "postISRCCD/%(runId)s/%(object)s/%(date)s/%(filter)s/postISRCCD-%(visit)d-%(ccd)02d.fits"
        composite: {
            info: {
                datasetType: composite_ExposureInfo
            }
        }
        python: "lsst.afw.image.ExposureF"
        persistable: "ExposureF"
        storage: "FitsStorage"
        level: "Ccd"
        tables: "raw"
        columns: "visit"
        columns: "ccd"
    }

    composite_ExposureInfo: {
        Template: "postISRCCD/%(runId)s/%(object)s/%(date)s/%(filter)s/postISRCCD-%(visit)d-%(ccd)02d.fits"
        composite: {
            wcs: {
                datasetType: wcs
            }
        }
        python: "lsst.afw.image.ExposureInfo"
        persistable: "ExposureInfo"
        storage: "FitsStorage"
        // I'm not sure if these fields are valid here?
        level: "Ccd"
        tables: "raw"
        columns: "visit"
        columns: "ccd"
    }

    wcs: {
        // would want to specify a different template?
        template: "calexp/%(runId)s/%(object)s/%(date)s/%(filter)s/%(tract)d/wcs-%(visit)d-%(ccd)02d.fits"
        python: "lsst.afw.image.Wcs"
        persistable: "Wcs"
        storage: "FitsStorage"
    }
}
```

Note; Exposure Class Hierarchy

Exposure is made up of many subclasses:

- Exposure
  - MaskedImageT
    - ImagePtr
    - MaskPtr
    - VariancePtr
  - ExposureInfo
    - WCS (ptr)
    - PSF (ptr)
    - Calib (ptr)
    - Detector (const ptr)
    - Polygon (const ptr) ‘valid polygon’
    - Filter (object)
    - PropertySet (ptr)
    - CoaddInputs (ptr)
    - ApCorrMap (ptr)

Assembler

Generic Assembler

We can build a generic assembler that can parse these policy hierarchies to build and persist composite datasets.

If a different assembler is required or desired, an assembler and disassembler that should be used can be indicated in the dataset’s composite section.

The generic algorithm proposal looks something like the following:

```
createObject(datasetType, dataId, classObj)
    components = {}
    if datasetType has 'composite':
        for name, componentInfo in datasetTypes.composite.iteritems():
            components[name] = createObject(val.componentInfo, dataId)
    obj = classObj()
    for name in components:
        setterName = (get setter from the policy else generate setter name)
        setter = getattr(obj, setterName)
        setter(components[name])
    return obj
```

Custom Assembler

If the policy specifies an assembler then instead of the generic assembler the specified assembler will be used to instantiate the object with components. A disassembler can also be specified

It is proposed that we can implement this feature initially using the generic assembler and not implement the custom assembler use until a need arises. This allows us to defer specifying the assembler and disassembler signatures until we have a concrete use case.

Init Order

The default assembler builds from the bottom up so that if a component object replaces a default part of a composite, the component will be applied after the composite has been initialized.

If it is possible for 2 component objects to overwrite the same data in a composite object then the order the setters are called in will become important. We could preserve the order the components are listed in the 'composite' section of the dataset policy to specify order. **TBD do we need this?**

TBD In the generic assembler, do we need to be able to interpose the root object class to a place other than last in the init sequence?

Duplicated Member Optimization

In our example it may be wasteful if the ExposureF constructor deserializes all of its members (and members-of-members, etc) out of one fits file, and then members are replaced.

TBD Do we need a way (now, or later) to optimize this?

Initial ideas/options:

- Pass member pointers (default to nullptr) to the constructor, if non-null, that component should not be loaded from disk (instead use the passed-in object)
    - This would require the setter/getter protocol to be adapted to a ‘constructor’ type protocol (maybe getters remain as-is, and ‘setter’ changes to ‘constructor’?)
- Pass flags to the constructor indicating what it should not (or should, if that makes any sense at all) build, implying a promise-to-set after the object has been constructed.
- Start changing objects that can be composites to always be composites. For example, maybe ExposureF would become a “pure composite” as described below.

Pure Composites

I'm using "pure composite" to mean an object that does not read or write member data directly. Instead it delegates all of its data I/O to member objects.

This kind of class would still have a dataset type entry in the policy but it would omit fields like 'template', 'storage', 'level', and fields used for database access.

It's possible that we could make a rule that all datasets are defined as either a pure composite or a non-composite. **TBD** needs research: determine if this would simplify implementation & use, and what the implication on butler-managed python objects would be.

Member Accessors in Python Objects

This proposal assumes that python types with component members will have setters and getters to set/get those members. It also proposes that composite object constructors/initializers should not deserialize component objects by default and/or should be able to receive component objects to prevent unneeded deserialization.

Setter and getter name derived from component name

It is easiest (in that we can have the least verbose policy) if we can derive the name from the component key. For example, this snippet of policy is taken from the example above:

```
postISRCCD: {
    python: "lsst.afw.image.ExposureF"
    composite: {
        info: {
            datasetType: composite_ExposureInfo
        }
    }
}
```

indicates that the ExposureF class has methods called setInfo and getInfo
that can be inferred (from 'info' at postISRCCD.composite.info).

TBD it could also indicate that the constructor has an argument called ‘info’? E.g. `ExposureF(..., info=None, ...)`

Setter and getter name specified in policy

If the setter and/or getter have a unique name they can be specified in the policy. For example:

```
postISRCCD: {
    python: "lsst.afw.image.ExposureF"
    composite: {
        info: {
            datasetType: composite_ExposureInfo
            setter: replaceInfo
            getter: fetchInfo
            initArg: exposureInfo
        }
    }
}
```

The above indicates that the ExposureF class has methods called replaceInfo and
fetchInfo that can be be used to access the ExposureInfo.

(if we implement constructor member passing, initArg indicates the argument name for passing the ExposureInfo to the constructor)

Object Registry With Component Metadata

I think in some cases we will have to maintain Butler & LSST pipeline data related dataset processing for objects in a repository. One natural place to do that is in the (sqlite) object registry.

Processing Stage Metadata Information

There is a requirement that objects be able to be somehow marked in a way to indicate what processing stage created or last modified that object. One example is Access Exposure With Updated WCS, where a `calexp` generated by single-visit processing must be findable in a way that is discrete from the `calexp` processed by `jointcal`. The Proposal is to add a qualifier on the dataset type name after a colon. For our example, the label "jointcal" can be appended to the dataset type name, e.g. `butler.get(calexp:jointcal, dataId={...})`.

One issue that arises is that the calexp generated by jointcal processing may contain components that were modified by jointcal processing *and* component s that were not modified by jointcal processing.

I think it will work to add information in the registry about what level of processing components were used when writing a composite dataset to the repository, when components came from a repository that was tagged with a processing-level identifier. The idea probably wants some fleshing out though. (some questions: do you add the processing stage only if it was requested when the object was created? do you add it any time the source repository is tagged with processing stage data?)