# Use Cases for Butler Composite Datasets

## Use cases from Jim Bosch

Here's the detailed list of use cases I can think of for composite datasets.  It's a bit more exhaustive that what I sent you before, so some of it may be new to you, but I don't think there's anything here that can't be met (at least initially) by your current design.  Some of these will require significant changes to Science Pipelines code as well, and hence aren't likely to happen soon.  When you put this into an RFC, you might want to include YAML snippets for how the composite datasets describe in each of these cases would be defined.  We should also invite others to come up with additional use cases in case there's functionality that would have to be added to support them.

### 1. Access Exposure With Updated WCS

In jointcal (and meas_mosaic) we generate a Wcs and (soon) Calib that supersede those included in the Exposure "calexp" dataset generated by single-visit processing. Downstream tasks should be able to access an updated Exposure as a different named dataset, via e.g. `butler.get('calexp: jointcal', ...)` or `butler.get('calexp', dataId={'version':'jointcal', ...})`

Because jointcal will be run at a tract level (or some other grouping of Exposures), and some "calexp" datasets will be associated with multiple tracts, the data ID of the new Wcs, the new Calib, and the updated Exposure will need to include a tract key even though the original "calexp" data ID does not.

#### Requirements to Satisfy This Use Case

- Composable and Decomposable & Pluggable
  - Use: allows the jointcal Wcs and the jointcal Calib to be stored individually, without storing data that would be duplicate with data in the exposure from single-visit processing.
- Datasets Findable by Processing Stage
  - This allows later processing stages to ask for a dataset that has been processed by a specific processing step, and would fail if a dataset is present but had not been processed to the extent expected by the current processing step.
  - When new Wcs and Calib components are written by the `jointcal` task, the output repository from jointcal should contain a definition of an Exposure dataset type that includes the pixels from the original calexp and the new Wcs and Calib components.

- Additional DataId Keys in Later Processing Stages
  - This allows the later processing stage(s) to search based on criteria that is derived in an intermediate processing step.

#### Pseudocode

Create a butler where the input repository contains the calexp datasets from single frame processing as Type 1 datasets, and the output repository will contain the jointcal processed images.

```
import lsst.daf.persistence as dafPersist
inputRepoArgs = dafPersist.RepositoryArgs(root='single_visit_processing')
outputRepoArgs = dafPersist.RepositoryArgs(root='jointcal_processing')
butler = dafPersistence.Butler(inputs=inputRepoArgs, outputRepoArgs)
```

The policy needs entries for the existing Type 1 calexp dataset, for Type 2 dataset getters to get components from the Type 1 dataset, and for Type 3 dataset to persist the composite dataset as component outputs.

Note that `calexp_wcs` and `calexp_calib` have the same `template` as `calexp`; this is because they come from the same persisted dataset. But, coming from the same dataset is is not required.

```
datasets: {
    calexp: {
        template:      "sci-results/%(run)d/%(camcol)d/%(filter)s/calexp/calexp-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:        "lsst.afw.image.ExposureF"
        persistable:   "ExposureF"
        storage:       "FitsStorage"
        level:         "None"
        tables:        raw
        tables:        raw_skyTile
    }
        # Type 2 getter for wcs in a Type 1 calexp
    calexp_wcs: {
        template:    "sci-results/%(run)d/%(camcol)d/%(filter)s/calexp/calexp-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:      "lsst.afw.image.Wcs"
        persistable: "Wsc"
        storage:     "FitsStorage"
    }
        # Type 2 getter for calib in a Type 1 calexp
    calexp_calib: {
        template:    "sci-results/%(run)d/%(camcol)d/%(filter)s/calexp/calexp-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:      "lsst.afw.image.Calib"
        persistable: "ignored"
        storage:     "FitsStorage"
    }
        # Type 1 datasets for components of the Type 3 jointcalexp
        joint_wcs: {
                template:    "wcs/.../filename.fits"
                python:      "lsst.afw.image.Wcs"
                persistable: "Wcs"
                storage:     "FitsStorage"
        }
        joint_calib: {
                template:    "wcs/.../filename.fits"
                python:      "lsst.afw.image.Calib"
                persistable: "Calib"
                storage:     "FitsStorage"
    }
    # Type 3 dataset definition for jointcalexp
    jointcalexp: {
        python:      "lsst.afw.image.ExposureF"
        composite: {
            calexp: {
                datasetType: "calexp"
                inputOnly: True # is this a good var name? meaning is do not write when deserializing.
            }
            wcs: {
                datasetType: "jointcal_wcs"
            }
            calib: {
                datasetType: "jointcal_calib"
            }
        }
        assembler: "lsst.mypackage.jointcal.JointcalAssembler"
        disassembler: "lsst.mypackage.jointcal.JointcalDisassembler"
    }
}
```

Assembler and Disassembler for `jointcalexp` are written and saved as specified by the policy. Assembler/Disassembler API is still TBD. Proposing:

Butler will get objects specified by components of type 2 and type 3 datasets and pass the group of those in a dict to the assembler with the component item key as the dict key. A class object of type indicated by the `python` field of the policy will be passed to the Assembler. If the object to return is a type 1 member of the composite, the Assembler may ignore the class object. Or, the assembler may create an instance via the class object and populate it with components from the component dict.

```
def JointcalAssembler(dataId, componentDict, classObj):
    # expects componentDict keys 'calexp', 'wcs' and 'calib', to contain Exposure, Wcs and Calib objects,
respectively.
    # in this case, load the 'base' object, and then overlay components
    exposure = componentDict['calexp']
    exposure.setWcs(componentDict['wcs'])
    exposure.setCalib(componentDict['calib'])
    return exposure
```

Calling butler.put to serialize the Exposure into the repository will decompose the Exposure via the disassembler.

This example shows only the updated `wcs` and `calib` being put into the butler's output repository. If they were the only items updated then the rest of the exposure should not need to be written; its data is unchanged from where it was located in the `single_visit_processing` repository.

```
def JointcalDisassembler(exposure, dataId, componentDict):
    componentDict['calexp'] = None
    componentDict['wcs'] = exposure.getWcs()
    componentDict['calib'] = exposure.getCalib()
```

Questions:

- What about putting a cached wcs (if the same wcs is shared among many exposures)? Should it be that the Disassembler puts a reference to the exposure in the componentDict, and when that type 1 dataset is to be written, butler will do a write-once-compare-same (requires redundant writes, but will throw away the 'same' file) operation? Or, it might work for the Butler cache to record that the object has already been written for a given dataset type + dataId, and skip the put if it knows it's been written.
- Do we want to require that all the datasets that should be in the component dict (as indicated by the policy) are there? We could do one of:
    - If a dataset type is not in the component dict, raise. Allow None to indicate that the dataset is input-only on this object type.
    - Silently ignore missing dataset types.

## Conversation

### Component Lookup by Processing Stage

*Unknown User (npease) asked:*

If new Wcs and Calib component datasets are written, but other component datasets are not replaced: is it important (or possible?) to specify which component datasets should be from `calexp.jointcal` and which are ok to have come from the single-visit `calexp`? I'm currently imagining a lookup algorithm for components:

```
butler.get(datasetType='calexp:jointcal', ...)
    for each component in composite:
        location = butler.map(datasetType, dataId)
        if location:
                        # I figure it will find locations for 'wcs' and 'calib'
                else:
            # There are lots of other possible components in MaskedImage
                        # and ExposureInfo (psf, detector, validPolygon, filter,
                        # coaddInputs, etc) What to do?
```

In the "What to do" case, would butler fall back to the next-previous processing step name? (how would it know?) Or, would butler remove 'stage' from the dataset type and search for component types in the repo-lookup order? If so, why bother with a stage qualification on the named composite? (if it can go ahead and use prior components if a component with the name does not exist)?

*Jim Bosch said:*

I believe there is always a strict ordering of stages, with the most recent stage always overriding previous stages. The stage lookup order should correspond to repo order, so at first glance it seems we could just drop the stage names entirely and rely on repo lookup order. As we discussed in Tucson, I believe that's a little dangerous in terms of provenance (especially in the case where some data units are run through all processing stages before other data units are run through any stages), which is why I'd prefer a design in which the stages are explicitly encoded in either the datasetType or the data ID.

All composite datasetType definitions would explicitly define which stages they'd get their components from, and once we have the ability to define datasetTypes within repos, we could have each stage add an alias that redirects datasetTypes without explicit stage names to the one with most recent stage. In the meantime, we can always just use the stage-qualified datasetTypes. I think that with this design we don't actually require a strict ordering of stages for lookup, which could be nice if I'm wrong about that always being true.

I don't think this is the only way to meet our requirements, but it seems safer and more explicit than more "automatic" approaches that would utilize the repo chain for component lookups or otherwise automatically assemble composites. That also makes it more fragile - several datasetType definitions would need to be updated when pipeline stages are reordered - but I think that's an acceptable cost for the safety. Other developers should probably weigh in on this.

### Additional DataId Keys

Unknown User (npease) asked:

- Does the key (and value) have to be in the policy's dataset type template, or could it be in registry metadata only? What about FITS metadata? (note that use of FITS is not always guaranteed)
- Is it possible to declare early that the dataId has a new expected key (perhaps we could keep a policy override in the repository?)
- Is it possible to know all the dataId keys at ingest time? (Then we could init the keys to 'null' whose values will be determined by an intermediate processing step, but not have to override policy data at an intermediate processing step.)

Jim Bosch said:

As for additional data ID keys, I believe these would always have to be present in the datasetType template of at least one of the updated components, and would always be necessary to load datasets after the stage that added the data ID key (because it would be necessary to disambiguate). For example, we might start with a single "calexp:sfm" dataset with (visit, ccd) keys. A later "jointcal" stage would define a "calexp.wcs:jointcal" dataset with (visit, ccd, tract) keys, implying that (visit, ccd) is no longer sufficient to identify a "calexp:jointcal". The additional data ID key would nearly always be "tract", actually - I can't think of a use case for anything else - and I don't think it will every be anything that will be present in a raw data registry. It is closely related to the spatial registry functionality we've asked for in the butler, however - and I could certainly imagine us asking for all "calexp:jointcal" datasets in a specific region on the sky, which then implies one or more tracts and allows the spatial query system to find all overlapping (visit, ccd) IDs.

## 2. Load Exposure Components Individually

Analysis code should be able to load components of an Exposure dataset individually without loading the full Exposure, using something like `butler.get('calexp.psf', ...)`. Whether the Exposure is stored as a single file on disk or as separate files for each component should be transparent to the user.

### Use Case Questions

- Can't the user code ask for a WCS dataset type? I think with the same dataId as would be used for the calexp?
  - Jim Bosch: I don't think we can get away with just a "wcs" dataset type, as there will be multiple WCSs (one for every Exposure dataset type).

### Requirements to Satisfy This Use Case

- Composable and Decomposable & Pluggable
  - Use: allows the Exposure and the Psf to be loaded separately.
- Component Access
  - Use: allows the component to be specified in terms of the composite calexp, but only returns the Psf of the calexp.

### Pseudocode - Type 2 and Type 3 calexp dataset

With a type 2 or type 3 dataset definition for calexp

```
    calexp_psf: {
        template:    "sci-results/%(run)d/%(camcol)d/%(filter)s/calexp/calexp-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:      "lsst.afw.detection.Psf"
        persistable: "Psf"
        storage:     "FitsStorage"
    }

    calexp: {
        python:      "lsst.afw.image.ExposureF"
        storage:     "FitsStorage"
        composite: {
            psf: {
                datasetType: "calexp_psf"
            }
            calib: {
                datasetType: "calib_psf" # (policy for this dataset type is not shown)
            }
        }
        assembler: "lsst.mypackage.CalexpAssembler"
        disassembler: "lsst.mypackage.CalexpDisassembler"
    }
```

The user calls `butler.get('calexp.psf', dataId={...})`. Butler looks up the dataset type definition for `calexp`, and in `composite` finds its component dataset type `psf`, which refers to the Type 1 dataset type `calexp_psf`. Butler gets the `Psf` object according to that dataset type definition and returns the object.

### Pseudocode - Type 1 calexp dataset

Given the policy:

```
    calexp: {
        template:    "sci-results/%(run)d/%(camcol)d/%(filter)s/calexp/calexp-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:      "lsst.afw.image.ExposureF"
        persistable: "ExposureF"
        storage:     "FitsStorage"
    }
```

Question

I can't think of how it's possible in a reliable way to get the Psf that an Exposure "would have" loaded (or any member object that a class would have instantiated in that class's deserializer) without actually running the class's deserializer. I do think we could instantiate the Exposure, infer that the name of the getter is `getPsf`, and return the result of that get operation. But this does not save us from having to instantiate an entire exposure class. Is this worth it, or is it better to just require that component loading requires a Type 2 or Type 3 dataset definition?

## 3. Individual Storage of Exposure & SourceCatalog Components, Plus Caching

Some components of an Exposure (including Wcs and Calib) are also conceptually associated with any SourceCatalog generated from that Exposure, and in the future we would like to actually attach these components to in-memory SourceCatalogs. When loading a SourceCatalog via the butler, these components should be read as well, whether they are persisted internally as part of the Exposure, as part of the SourceCatalog, or independently. This must not involve reading the full Exposure, as this would be much more expensive. Ideally when both a SourceCatalog and its associated Exposure are loaded through the butler the component shared_ptrs attached to each would point to the same underlying object, but this is probably an optimization, not a requirement.

### Requirements to Satisfy This Use Case

- Composable and Decomposable & Pluggable
    - Use: given plugin code for building a SourceCatalog and an Exposure from component datasets, the Butler can assemble each of these composite objects from their component datasets.
- Caching
    - Use: if a component has already been loaded and is needed again as an object of another dataset, the butler can install a pointer/ref to the already-constructed component objects in the composite objects instead of creating another instance of the same object.

### Pseudocode

Start with an input repository that has `SourceCatalogs`, one (or more?) `Wcs`, and `Calib`s stored separately.

#### By Setting/Getting Components or Monkey Patching

Use a policy that describes a dataset type that describes the composite dataset type:

```
 datasets: {
    icSrc: {
        template:      "sci-results/%(run)d/%(camcol)d/%(filter)s/icSrc/icSrc-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:        "lsst.afw.table.SourceCatalog"
        persistable:   "ignored"
        storage:       "FitsCatalogStorage"
        tables:        raw
        tables:        raw_skyTile
    }
        wcs: {
                template:    "wcs/.../filename.fits"
                python:      "lsst.afw.image.Wcs"
                persistable: "Wcs"
                storage:     "FitsStorage"
        }

        calib: {
                template:    "wcs/.../filename.fits"
                python:      "lsst.afw.image.Wcs"
                persistable: "Calib"
                storage:     "FitsStorage"
    }
    extended_icSrc: {
        python: "lsst.afw.table.SourceCatalog"
        composite: {
            icSrc: {
                datasetType: "icSrc"
            }
            wcs: {
                datasettype: "wcs"
            }
            calib: {
                calib: "calib"
            }
        }
        assembler: "lsst.mypackage.extended_icSrc_assembler"
        disassembler: "lsst.mypackage.extended_icSrc_disassembler"
    }
}
```

The assembler and disassembler could like this & do monkey patching the SourceCatalog

```
def extended_icSrc_assembler(dataId, componentDict, classObj):
    # in this case, load the 'base' object, and then add in components as a monkey patch
    srcCat = componentDict['icSrc']
    srcCat.wcs = componentDict['wcs']
    srcCat.calib = componentDict['calib']
    return srcCat

def extended_icSrc_disassembler(srcCat, dataId, componentDict):
    componentDict['icSrc'] = srcCat
    componentDict['wsc'] = srcCat.wsc
    componentDict['calib'] = srcCat.calib
```

Or if the SourceCatalog class was extended to have setters & getters then the setters could be called by the assembler and the getters by the disassembler.

```
def extended_icSrc_assembler(dataId, componentDict, classObj):
    # in this case, load the 'base' object, and then add in components as a monkey patch
    srcCat = componentDict['icSrc']
    srcCat.setWcs(componentDict['wcs'])
    srcCat.setCalib(componentDict['calib'])
    return srcCat

def extended_icSrc_disassembler(srcCat, dataId, componentDict):
    componentDict['icSrc'] = srcCat
    componentDict['wsc'] = srcCat.getWsc()
    componentDict['calib'] = srcCat.getCalib()
```

### By Pure-Composite Container Class

Another way would be to write a python type that contains source catalog, `Wcs,` and `Calib` and use the generic assembler.

```
class SourceCatalogWithInfo:
    def __init__(self):
        """no-op constructor"""
        pass

    # set all the component data individually:
    def setSourceCatalog(self, sourceCatalog):
        self.sourceCatalog = sourceCatalog
    def getSourceCatalog(self):
        return self.sourceCatalog
    def setWcs(wcs):
        self.wcs = wcs
    def getWcs(wcs):
        return self.wcs
    def setCalib(calib):
        self.calib = calib
    def getCalib(calib):
        return self.calib
```

Assembler & Disassembler:

```
def extended_icSrc_assembler(dataId, componentDict, classObj):
    # in this case, load the 'base' object, and then add in components as a monkey patch
    srcCatEx = classObj()
        srcCatEx.setSourceCatalog(componentDict['icSrc'])
    srcCatEx.setWcs(componentDict['wcs'])
    srcCatEx.setCalib(componentDict['calib'])
    return srcCatEx

def extended_icSrc_disassembler(srcCatEx, dataId, componentDict):
    componentDict['icSrc'] = srcCatEx.getSourceCatalog()
    componentDict['wsc'] = srcCatEx.getWsc()
    componentDict['calib'] = srcCatEx.getCalib()
```

Modify the policy to use the new type

```
    extended_icSrc: {
        python: "lsst.afw.table.SourceCatalogWithInfo"
        composite: {
            icSrc: {
                datasetType: "icSrc"
            }
            wcs: {
                datasetType: "wcs"
            }
            calib: {
                calib: "calib"
            }
        }
    }
}
```

### Use

```
import lsst.daf.persistence as dafPersist
butler = dafPersist.butler(inputs="repo/A/path", outputs="repo/B/path")
dataId = {...}
srcCatObj = butler.get('icSrc', dataId)
SourceCatalogWithInfo = srcCatEx()
srcCatEx.setSourceCatalog(srcCatObj)
srcCatEx.setWcs(butler.get('wcs', dataId))
srcCatEx.setCalib(butler.get('calib', dataId))
butler.put('extended_icSrc', dataId)
# note, none of the contained objects were changed. I guess they should still be written though?
# we've talked some about the object cache, and how to indicate that an object should const (ok to share, won't
change)
# and how to indicate that it's being loaded with the intent to modify (should not be shared).
# in the const case, it maybe should not get written if it already exists in an input (parent) repo as a
persisted dataset?

# ==== later =====


butler = dafPersist.butler(inputs="repo/B/path", outputs="repo/C/path")
reloadedSrcCatEx = butler.get('extended_icSrc, dataId)
# and then use the source catalog with wcs and calib
```

## 4. Store Processed-Exposure Dataset Components Separately

When processing coadds, the first stage (detection) modifies only the background (a lightweight object that can be added or subtracted from the image) and a single mask plane, and no subsequent steps modify the coadd Exposure object at all. We would like to define Exposure datasets that represent both the coadd prior to detection and the coadd after detection, without duplicating on-disk the Exposure components that are shared.

Here the delta between the two images are currently represented as operations beyond just get/set (add/subtract for backgrounds, bitwise AND/OR/NOT for masks), but modifying Science Pipelines code to treat these as get/set (by adding a Background component to Exposure and treating Mask planes more atomically) may be an acceptable solution.

Requirements to Satisfy This Use Case

- Composable and Decomposable & Pluggable
    - Use: The butler plugin for building the coadd needs to be written so that replaceable component objects are fetched individually. As newer versions of component objects are created they are 'put' into later/newer repositories. This way they they mask earlier versions of objects (and/or do they need to be findable by Processing Stage similar to how it's described in Access Exposure With Updated WCS?).

### Pseudocode

Note, Jim says "We haven't yet implemented attaching background objects to Exposures yet, but it's been our to-do list for a long time.Currently our background class is pure-Python."

The background class is `lsst.afw.math.BackgroundList`

Implement a container class for Exposure and Background:

```
class ExposureWithBackground:
    def __init__(self):
        pass
    def setExposure(self, exposure):
        self.exposure = exposure
    def getExposure(self):
        return self.exposure
    def setBackground(self, background):
        self.background = background
    def getBackground(self):
        return self.background
```

create a policy that describes the dataset type that will use ExposureWithBackground, with a dataset type for before-detection data, and a type for after-detection data.

```
datasets: {
    calexpBackground: {
        template:       "sci-results/%(run)d/%(camcol)d/%(filter)s/calexp/bkgd-calexp-%(run)06d-%(filter)s%
(camcol)d-%(field)04d.fits"
        python:        "lsst.afw.math.BackgroundList"
        persistable:   "PurePythonClass"
        storage:       "FitsCatalogStorage"
        tables:        raw
        tables:        raw_skyTile
    }
    calexp: {
        template:      "sci-results/%(run)d/%(camcol)d/%(filter)s/calexp/calexp-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:      "lsst.afw.image.ExposureF"
        persistable: "ExposureF"
        storage:     "FitsStorage"
        level:       "None"
        tables:      raw
        tables:      raw_skyTile
    }
    coadd_detection: {
        python: "lsst.afw.image.ExposureWithBackground"
        composite: {
            background: {
                datasetType: calexpBackground
            }
            exposure: {
                datasetType: calexp
            }
        }
        assembler: "lsst.mypackage.ExposureWithBackgroundAssembler"
        disassembler: "lsst.mypackage.ExposureWithBackgroundDisassembler"
    }
    coadd_postdetect {
        python: "lsst.afw.image.ExposureWithBackground"
        composite: {
            background: {
                datasetType: calexpBackground
            }
            exposure: {
                datasetType: calexp
                inputOnly = True
            }
        }
        assembler: "lsst.mypackage.ExposureWithBackgroundAssembler"
        disassembler: "lsst.mypackage.ExposureWithBackgroundDisassembler"
    }
}
```

write assemblers and disassemblers

```
def ExposureWithBackgroundAssembler(dataId, componentDict, classObj):
    exposure = componentDict['exposure']
    exposure.setBackground(componentDict['background'])
    return exposure
def ExposureWithBackgroundDisassembler(obj, dataId, componentDict):
    componentDict['exposure'] = obj.exposure
    componentDict['background'] = obj.background
```

Use:

```
#Create a butler with the pre-detection data as inputs and will put the post-detection data for output
import lsst.daf.persistence as dafPersist
butler = dafPersist.Butler(input="datasets/A", output="datasets/B")
dataId = {...}
exp = butler.get("coadd_detection", dataId)
exp.runDetectionProcessing()
# this put will write both the background and the exposure to the repository at "datasets/B"
butler.put("coadd_detection", dataId)
# later...
butler = dafPersist.Butler(input="datasets/B", outputs="datasets/C")
exp = butler.get("coadd_postdetect", dataId)
# do not modify the exposure object - it will not get written!)
exp.performOperationsThatDoNotChangeTheExposure()
# this will write the background to the repo at "datasets/C" but not the exposure.
butler.put("coadd_postdetect", dataId)
```

## 5. Composite Object Tables

Object tables are currently written as a set of several SourceCatalogs datasets that column-partition the full table. It should be possible to define an Object dataset that combines these SourceCatalog datasets in the column direction. This will also require changes to the table library as well as the butler and may not be implemented soon, but should be considered in the composite dataset design.

### Requirements to Satisfy This Use Case

- Composable and Decomposable & Pluggable
    - Use: requires that a Butler plugin can be registered for a dataset type name that handles building the SourceCatalog in the desired way.

### Pseudocode

Create a policy. In this case the composite object is a type 2 object, and should not have a disassembler because it should not be doing a lossy write to component datasets. If needed, a type 1 or type 3 dataset type can be defined. The type 3 would write a reduced-data version of the components.

```
datasets: {
    srccat_blue: {
        # note, this hard-codes the blue filter by changing the beginning of the template from
        # "sci-results/%(run)d/%(camcol)d/%(filter)s/...
        # to
        # "sci-results/%(run)d/%(camcol)d/b/...
        template:    "sci-results/%(run)d/%(camcol)d/b/icSrc/icSrc-%(run)06d-%(filter)s%(camcol)d-%(field)04d.
fits"
        python:        "lsst.afw.table.SourceCatalog"
        persistable:   "ignored"
        storage:       "FitsCatalogStorage"
        tables:        raw
        tables:        raw_skyTile
    }
    srccat_green: {
        # note, this hard codes green
        template:    "sci-results/%(run)d/%(camcol)d/g/icSrc/icSrc-%(run)06d-%(filter)s%(camcol)d-%(field)04d.
fits"
        python:        "lsst.afw.table.SourceCatalog"
        persistable:   "ignored"
        storage:       "FitsCatalogStorage"
        tables:        raw
        tables:        raw_skyTile
    }
    srccat_bg: {
        python: "lsst.afw.table.SourceCatalog"
        composite: {
            blue: {
                datasetType: "srccat_blue"
            }
            green: {
                datasetType: "srccat_green"
            }
        }
        assembler: 'lsst.mypackage.srccat_bg_assembler'
    }
}
```

Which makes a case for dataset inheritance ('genre'?)
I haven't thought hard about how to indicate policy inheritance or the consequences of this approach,
but the idea here is much like `python: Class(base class)`

```
datasets: {
    srccat: {
        template:    "sci-results/%(run)d/%(camcol)d/%(filter)s/icSrc/icSrc-%(run)06d-%(filter)s%(camcol)d-%
(field)04d.fits"
        python:        "lsst.afw.table.SourceCatalog"
        persistable:   "ignored"
        storage:       "FitsCatalogStorage"
        tables:        raw
        tables:        raw_skyTile
    }
    srccat_blue(srccat): {
        template:    "sci-results/%(run)d/%(camcol)d/b/icSrc/icSrc-%(run)06d-%(filter)s%(camcol)d-%(field)04d.
fits"
    }
    srccat_green(srccat): {
        template:    "sci-results/%(run)d/%(camcol)d/g/icSrc/icSrc-%(run)06d-%(filter)s%(camcol)d-%(field)04d.
fits"
    }
    srccat_bg: {
        python: "lsst.afw.table.SourceCatalog"
        composite: {
            blue: {
                datasetType: "srccat_blue"
            }
            green: {
                datasetType: "srccat_green"
            }
        }
        assembler: 'lsst.mypackage.srccat_bg_assembler'
    }

    post_assembled_bg_srccat: {
        # note the template replaces "%(filter)s" with "bg"
        template: "sci-results/%(run)d/%(camcol)d/bg/icSrc/icSrc-%(run)06d-%(filter)s%(camcol)d-%(field)04d.
fits"
        python: "lsst.afw.table.SourceCatalog"
        storage: "FitsCatalogStorage"
}
```

Assembler

```
def srccat_bg_assembler(dataId, componentDict, classObj):
    # my understanding, per the example, is that there is no function to combine these datasets in the way
    # that the example wants to do it, so we'll just use a 'fake' function called 'doAssembly'.
    obj = classObj()
    obj = doAssembly(obj, componentDict['blue'], componentDict['green']
    return obj
```

Create a butler, get the composite object, put the object as a type 1.

```
import lsst.daf.persistence as dafPersist
butler = dafPersist.Butler(input='path/to/files', output='path/to/put')
dataId = {...}
bgSrcCat = butler.get("srccat_bg", dataId)
butler.put(bgSrcCat, "post_assembled_bg_srccat", dataId)
```

## 6. Aggregates and Aggregate Dataset Type Name

Most catalog datasets and some image datasets can be trivially aggregated according to nested data IDs. For example, a catalog dataset for a tract (or visit) can be generated simply by combining all catalogs for the patches within that tract (ccd) in the row direction. Access to these aggregate datasets should probably use a different name than their constituents, to avoid confusion with other interpretations of partial data IDs.

Aggregating datasets may include combining structured metadata or components that they share (which should already be consistent but may be duplicated).

Requirements to Satisfy This Use Case

- Composable and Decomposable & Pluggable
    - Use: for the dataset name specified for the aggregate case there will be a plugin that knows how to combine (or knows how to call the constructor in a way that will cause it to combine) the datasets into a single aggregate object.

### Pseudocode

Question: is there a case where the aggregate composite should be persisted as a whole? Probably it would get written directly as a type 1 dataset? Or there would be a a disassembler (not shown) that knows how to break apart the aggregate deepCoadd and pass it back to be persisted as `deepCoadd_src`

Create a policy to describe the catelog and the aggregate catalog.
The aggregate needs a flag in the component whose entry in the componentDict (passed to the assembler) should have more than 1 instance of a component. Right now I'm using the keyword "subset", and if it's set to True, butler will put a list in the componentDict for that dataset type.

```
datasets: {
    deepCoadd_src: {
        template:      "deepCoadd-results/%(filter)s/%(tract)d/%(patch)s/src-%(filter)s-%(tract)d-%(patch)s.
fits"
        python:        "lsst.afw.table.SourceCatalog"
        persistable:   "ignored"
        storage:       "FitsCatalogStorage"
        tables:        raw
        tables:        raw_skyTile
    }
    deepCoadd_src_aggregate: {
        python: "lsst.afw.table.SourceCatalog"
        composite: {
            deepCoaddSrc: {
                datasetType: "deepCoadd_src"
                subset: True
            }
        }
        assembler: "lsst.mypackage.AssembleAggregateDeepCoadd"
    }
}
```

Assembler

```
def AssembleAggregateDeepCoadd(dataId, componentDict, classObj):
    baseObject = classObj()
    for obj in componentDict['deepCoaddSrc']:
        ret += obj
    return baseObject
```

Use

```
from lsst.daf.persistence import dafPersist
butler = dafPersist.butler(inputs="my/input/dir")
dataId {'filter':g} # note this leaves out tract and patch
sourceCat = butler.get("deepCoadd_src_aggregate", dataId)
# ...use the sourceCat
```

## 7. Access to Precursor Datasets (via Persisted DataId)

Coadd Exposures require access to some of the components of the sensor-level Exposures that were used to construct them. These are currently persisted (duplicated) directly within the coadd Exposure dataset, but we could also use butler composite datasets to assemble these from the original files, which would require support for composite datasets in which the data IDs for the components must themselves be unpersisted. There are potential disadvantages to normalizing persistence in this case (many small files are less convenient and sometimes less performant) that may preclude actually making this change even if the butler supported it, however.

Requirements to Satisfy This Use Case

- Associated Object DataIds Persisted With Dataset
- Component DataId Extraction From Dataset Metadata

Use: Storing dataset+dataId lookup information in a registry or in a persisted object's metadata will allow butler to find component objects, even when the dataId passed to a butler function provides enough information for a composite to be found but not for its components to be found.

Other Use Cases

## 8. Subsectioning Repositories Along the Dataset Axis (from Simon Krughoff)

The `calexp` is a composite dataset comprising a `MaskedImage` and other things like the Psf, Wcs, and Calib objects. The `src` is effectively a composite dataset (though it is not stored this way) because it needs the Calib object from the calexp in order to put the instrumental fluxes in a calibrated system.

We know of cases where we want to create a new repository with just the src catalogs for all the dataIds because the images are so large. This implies that composite datasets should be able to share components and be able to (un)persist the shared components independently.

Notes Regarding Item 8

`calexp` is a dataset type, typically associated with an `ExposureF` or `ExposureD` object type. `src` is a dataset type, typically associated with a `SourceCatalog` object type. Both contain a `calib` object which can be a common component of the two objects.

Requirements to Satisfy This Use Case

- Composable and Decomposable
- Distributed

Use: when performing a get operation, while searching for components, butler can look in more than one input repository for the component objects and assemble them assuming a unique match was found for each component object's dataset type and the dataId provided.

## 9. WCS Stored Separately from Exposure, and Replacing Existing Images In Exposure (from Jim Bosch

(This is the original example from Jim Bosch)
*Exposure consists of a MaskedImage and an ExposureInfo; the former contains three separate images, and the latter is just a bucket of more complex objects held by std::shared_ptr (Psf, Wcs, Calib, …). All of these are currently persisted together (i.e. it's a type 1 composite). We already have clear use cases for:*

- *Defining a new dataset that takes an existing Exposure dataset and replaces its Wcs with one persisted elsewhere.*
- *Defining a new dataset that takes an existing Exposure dataset and replaces one or two of its images with one(s) persisted elsewhere. Single-image replacements (specifically mask replacements) are probably the most common.*

*It might be useful to start with just these two use cases, or it might be useful to start by just decomposing all of Exposure into its constituents.*

Requirements to Satisfy This Use Case

- Composable and Decomposable
- Distributed
- Policy In Repo

## 10. Full Camera Visit Metadata Shared by All the CCD Exposures Associated With That Visit (from John Parejko )

Our *Exposure* and *SourceCatalog* objects are designed to hold the data from one CCD. Much of the exposure metadata (e.g. *ExposureInfo*) is visit-wide (e.g. observatory information, telescope boresight, some components of the WCS) and we may often want to work with full focal plane visits (e.g. initial astrometric fitting to prevent single CCD catastrophic failures, jointcal's future full focal plane fits). Having a "VisitExposure" type of object would help to manage the data, and would let questions about caching exposures, catalogs and metadata across the visit be managed at the butler level.