Butler Major Design Questions, 2019-05-28

This page describes various design questions (open issues, more concrete proposals, some vague worries, and everything in between) that affect the Gen3 Butler's overall architecture or otherwise merit gathering input from others.

Nested Transactions / Savepoints

Our design relies on Registry and Datastore state remaining consistent, and we utilize transactions to ensure this at the smallest possible unit: a single dataset. We also have bulk operations that involve many datasets that we would at least *like* to perform a atomically (it is unclear to what extent we have a requirement for atomic bulk operations). And in SQLite, the file-locking model we currently use strongly favors wrapping bulk operations in transactions for performance reasons (at least when only one process is writing to the DB) - so we wrap perhaps more operations than we need to in bulk transactions.

These use cases could be met with the the nesting of transactions implemented entirely in Python, by making interior transaction contexts no-ops when an outer transaction context is already active. This approach fails when we rely on catching and recovering from expected failures (usually foreign key or unique constraint violations) in order to implement desired behavior. Many registry operations currently use this approach to provide APIs that only insert records when equivalent records do not already exist. Using pure-Python transaction nesting in these contexts doesn't work; the constraint violation we would like to ignore immediately triggers the rollback of the outermost transaction, before we even have an opportunity to catch the corresponding Python exception. As a result, we are forced to instead implement nested transactions inside the database using savepoints.

That results in way too many savepoints during bulk operations. There are two ways to address this problem:

- If we add vectorized versions for all interfaces that are executed in bulk, the frequency of savepoints will decrease dramatically (~one per table, instead of one per record).
- If we can adopt conflict-resolution approaches that do not require immediate reporting of errors, we may be able to move the nesting implementation to Python. We may need to do this anyway in order to support vectorized operations with this behavior.

Generalized Observations

The current dimension schema defines visits and exposures as distinct concepts with a fairly rigid relationship defined at observation-time (or at least ingest-time). This now seems inadequate for a number of reasons:

- We would like to have the flexibility to redefine what exposures constitute a visit after ingest within a repo without invalidating datasets processed with previous visit definitions.
- Raw calibration frames are also taken in observational modes that define groups that are relevant for processing, but we have no way to capture these in the current data model. At least some of these relationships will be redefined much more often than science visit definitions; in some cases those relationships may be provided every time certain pipelines are run (which also begs the question of whether those particular relationships should be put into the database at all).
- For science observations taken in single snaps (either LSST "Alternate Standard Visits" or images from other cameras), the existence of visit and exposure as distinct entities is an unnecessary and confusing complication.

The CAOM2 data model defines an observation concept that may be a better fit for these use cases, because it is both more general and self nesting: an observation can contain other observations. A dimension schema in which observations replace both visits and exposures can't be tied directly to PipelineTasks or DatasetTypes, however, because these do map to only certain types of observations (e.g. the raws input to ISR are strictly non-composite observations, and the calexps input to coaddition must be on-sky science frames). So while adopting something like the CAOM2 observation concept as a generalization of our visit and exposure concepts provides a unified way to group things, we still need to work out a system for enumerating the different kinds of observations that can exist. This would require extensions to the dimensions framework itself, as we'd need to add some way to define a `DatasetType` such that only some entries for a certain dimension are valid for that `DatasetType`.

Making the mapping between visits and exposures more flexible also requires a way to represent a complete (or at least self-consistent) set of such mappings in the database, as well as a way to choose between them when launching pipelines. This could be done within the existing dimension framework by adding a "system of visits" dimension that would be a required dependency (much like the instrument) of the visit dimension. On the database side, that would also involve a new join table with a definition something like this:

```
CREATE TABLE exposure_visit_mapping AS (
    instrument VARCHAR,
    exposure_id INTEGER,
    visit_id INTEGER,
    mapping_system VARCHAR,
    FOREIGN KEY (instrument, exposure_id) REFERENCES observation (
        instrument, observation_id
    ),
    FOREIGN KEY (instrument, visit_id) REFERENCES observation (
        instrument, observation_id
    )
);
```

which would essentially only ever be called with queries that include

```
WHERE mapping_system = :some_literal_string:
```

This raises the question of whether different mapping systems belong in the same table at all, or whether it would be better to just give each system its own table (with all such tables having the same structure, indices, constraints, etc). Butler Python client code would select which such table to use via configuration or other user input; because that selection needs to be per-client (not per-schema), it cannot be implemented via (non-temporary) views. Using different tables for different mapping systems would thus represent a small gap in our goal of providing a consistent high-level (but read-only) SQL interface as well as a Python interface, because direct SQL access would have to manually use the appropriate mapping table instead of having it selected via configuration as in the Python interfaces. Multi-user registries may present a much bigger challenge to that goal, and we decide that providing such an interface is not practical, it may be worth considering whether other dimensions (such as instruments and skymaps) should also be represented as partitions of other dimensions (i.e. observations and tracts) instead of additional fields that can be used to filter the current, more monolithic, dimension tables.

obs-specialized Dimensions

One of the initial simplifying assumptions and advantages of the Gen3 Butler (from the perspective of writing instrument-generic pipeline code) was that we could conform all supported cameras to a common data model and nomenclature, noting that HSC, DECam, and CFHT Megacam were already largely consistent in this respect. Unfortunately, LSST has chosen significantly different terminology, grouping, and identifiers for both detectors and observations, so we now have to either awkwardly map LSST data to the existing system, (further) break backwards compatibility with the precursor instruments we already support by awkwardly mapping them to the LSST system, or add sufficient redundancy to the data model and indirection in Butler code to support a superset of the LSST and HSC/DECam/CFHT systems.

The latter approach is actually probably the only viable one; using the generic/precursor system internally with an LSST facade probably still lets the generic nomenclature "leak out" too much to meet obligations to other subsystems, and changing how we handle precursor data is too distruptive to both DM and (for HSC in particular) external partners. Implementing this approach presents some challenges, because we some aspects of the system don't really work well with that kind of redundancy:

- database tables can only have one primary key (which will also be visible in foreign key fields in other tables);
- instrument-generic pipeline code will either need one system of identifiers to use internally (for logging, provenance, etc.) or additional complexity to allow them to query what system to use.

Redundancy in these identifiers also has efficiency implications, in terms of the sizes of database tables and their associated in-memory Python objects (such as data IDs).

There is little to do here but hammer out the details of a compromise/superset identifier system that can work for all cameras.

Vectorized Autoincrement Inserts

Raw ingest and other data-repository-bootstrapping tasks currently use single-row INSERT statements. One of the challenges in providing vectorized versions of these operations applies only to tables with autoincrement primary keys (`dataset` in particular), in that we need those database-generated primary key values returned to Python so they can be used in subsequent inserts into related tables.

Obtaining ID values in advance by explicitly querying a sequence may provide a database-generic way to do this, but this may be less efficient, as it requires at least two distinct database operations separated by Python code for a single logical operation.

An alternative for SQLite would be to rely on SQLAlchemy support for retrieving the last inserted ID and number of inserted rows, and assume that the inserted IDs are continuous; this should be guaranteed by our use of exclusive locks (though this still feels a bit like a hack).

Oracle does have syntax for returning inserted values from an INSERT statement more directly, but it isn't supported by SQLAlchemy for reasons I don't fully understand (as it does provide support for similar functionality in other databases, while seeming to claim that Oracle itself has no such functionality); it may just be a gap in SQLAlchemy Oracle support that we might want to consider contributing upstream.

Collection Uniqueness Violations

Another challenge with bulk inserts into the `dataset` table is the handling of errors, particularly violations of the unique constraint (on `dataset_collection.dataset_ref_hash` and `dataset_collection.collection`) that prevents us from having multiple datasets with the same dataset type and dimensions in a single collection.

Note that this isn't the only unique constraint on that table - the primary key is compound, on `dataset_id` and `collection`, and inserts that violate the primary key constraint (which will always also violate the other one, because a particular `dataset_id` always implies the same `dataset_ref_hash`) should be silently ignored. Doing so will complicate the implementation of any error-handling approach for the non-primary-key constraint.

But the first question is how we *want* such errors to be handled, particularly in raw ingest or bulk transfers between repositories. There are a few options:

1. Any conflict represents a logic bug in higher-level code, and the entire operation (and possibly other operations in the same transaction) should be rolled back.
2. Conflicts represent problems with individual datasets that are not representative of the bulk operation as a whole; non-conflicting rows should be inserted, but with problems reported at least via log messages and ideally programmatically (e.g. by returning the `DatasetRef` objects, filenames, etc. for rows that were not inserted).
3. Conflicts are expected, and new datasets should take precedence over old ones, and hence old datasets should be removed from the collection so new ones can be inserted (which can be expressed as an update on the conflicting rows).
4. Conflicts are expected, and old datasets should take precedence over new ones, and hence conflicting inserts should be silently ignored.

Implementing just (1) is straightforward in any database: while the SQL-standard behavior for errors is to immediately roll back the operation that triggered the violation, but not the entire transaction, it's trivial to trigger a complete rollback at the Python level, and this is naturally what SQLAlchemy will do. What's tricky is combining this with the desire to ignore primary key uniqueness violation. SQLite provides custom syntax for requesting this behavior directly in the DDL for the constraint (`ON CONFLICT ROLLBACK`), though it's unclear (see DM-17419) whether we can *also* use DDL-level `ON CONFLICT IGNORE` on the primary key constraint in a way that guarantees that primary-key uniqueness violations trigger the ignore before the other constraint violation triggers a rollback. But SQLite also provides a way to specify conflict resolution as part of the insert, and while I don't know which of those takes precedence when both are provided, it seems likely that this is well-defined, and hence we can use a combination to get the behavior we want. Oracle has `MERGE` syntax that I understand to be similar to the latter, but it's not clear (to me, at least) whether that can be used to solve this problem. I don't know if it also has any DDL-level way to indicate how to resolve conflicts.

I don't see a good way to do (2) in SQLite. One possibility involves retrying (in Python) bulk operations multiple times with DDL-level `ON CONFLICT FAIL`, which will allow previous insertions to succeed but return as soon as it sees the first conflict. Another involves using `ON CONFLICT IGNORE`, but following up with `SELECT` queries to try to identify which insertions were not successful. Both of these seem fragile, and the latter in particular may be impossible to make safely concurrent. It doesn't seem like `MERGE` directly solves this problem in Oracle, either, though it does provide error logging controls that might provide a way to retreive the desired information about failed records, if not necessarily in the form we'd find most natural.

I think approaches (3) and (4) are the most straightforward - in SQLite (and PostgreSQL, from which this syntax is borrowed) they can be handled with queries like (3):

```
INSERT INTO dataset_collection ...
    ON CONFLICT (dataset_ref_hash, collection) DO UPDATE
        SET dataset_id=excluded.dataset_id
        WHERE dataset_id != excluded.dataset_id;
```

or (4):

```
INSERT INTO dataset_collection ...
    ON CONFLICT (dataset_ref_hash, collection) DO NOTHING;
```

I'm fairly confident there are Oracle equivalents to these using `MERGE`, as it seems slightly *more* flexible than the `ON CONFLICT` syntax, but haven't tried to work out what they'd be.

Given the disparities in (seeming) ease of implementation, it may be best to start by supporting only the error-handling approaches we can support easily, and waiting to see if there are real use cases for the more difficult approaches. I have not yet been able to identify any hard requirements that say we need any particular approach, but there may still be a large gap in user experience between them for various operations (having a multi-day raw ingest or transfer job rolled back due to some kind of irrelevant user error near the end would be quite frustrating, for example).

Table Indirection and Multi-User Registries

An original design goal for the Gen3 `Butler` was to provide both a Python interface and a high-level (but read-only) SQL interface that could be used directly across different database backends.

We have been able to achieve this so far, but we have also tried to avoid relying on the existence of a textual SQL interface in supporting key use cases. At the same time, multi-user registries will force us to rely on in-database indirection techniques (namely views) much more than we have thus far, because many logical butler tables will be implemented as union views of (e.g.) personal, shared, and offical versions of those tables (which could come from different Oracle schemas or SQLite database files).

A key question here is whether the definition of the high-level logical schema as a particular concrete schema/database chain is specific to a particular Butler client process and corresponding database session, or persistent in the last concrete (i.e. user) schema/database.

Given Oracle's lack of support for temporary views, it is probably impossible to support a high-level SQL interface in the per-client indirection model (unless there is a completely different Oracle feature I'm unaware of that we could use instead). It's worth emphasizing that this is *only* a problem for the high-level SQL interface; our Python APIs could easily support logical views by constructing union queries on-the-fly. And even with temporary views (which we do have in SQLite and PostgreSQL), direct SQL access would still have to go through the Python interface that sets up those views according to Butler client configuration, rather than a native database terminal application or other client.

Dropping the high-level SQL interface would probably eventually demand more from a custom query language/system. This could be done by expanding the extremely simple expression system that is currently used in `PipelineTask` preflight to include some support for joins and output specifications (i.e. right now it's just `WHERE` clauses; it may need to grow some way of specifying `SELECT` and `FROM` as well). That would result in much more concise queries for common cases (given our knowledge of the schema and its relationships, we could probably save users from ever having to write out the equivalent of an `ON` expression), but expanding it to the full generality of SQL would obviously be a poor use of developer time. Using a true SQL parser (ideally a third party library, or an in-house one developed for the LSP?) just to translate logical table names into union subqueries before passing them along would be another option. And, of course, it's not clear we really do need a generic SQL interface at the logical-table level, as direct SQL access to the underlying concrete tables would of course still be available to administrators, operators, and power users.

Using persistent views to implement table indirection would keep the goal of a high-level SQL interface alive, but it brings other complications. Unless users are given multiple personal schemas, they would have to drop and add those views whenever they wish to modify the chain of schemas that defines their logical repository. That would make comparisons between repositories with incompatible dependencies (different data releases may fall into this category) at least quite difficult. It may also make it impossible for users to launch a long-running job against one logical repository while (e.g.) interactively querying another. Finally, using views inside the database forces optimizing transformations that involve them to be made automatically by the query optimizer, instead of manually by Python code. While we of course want to offload whatever optimization we can to the database, limiting our options for manual tuning is also a concern, especially given our use of extremely large and complex queries in preflight.

A hybrid approach is also worth considering: we could avoid any use of these views (and possibly even other views unrelated to multi-user registries) in any queries generated by Python code, but still provide tools to explicitly create them for use in direct SQL queries. A user could then have multiple Butler-dependent Python jobs running at once, on different chains of schemas, without conflict, as well as a set of views for one schema chain to support (presumably mostly interactive or at least ad-hoc) direct SQL queries.

Partitioning the Dataset Table

The vast majority of Butler queries against the `dataset` table include a WHERE clause that limits the results to a single, literal value of the `dataset_type_name` field. Any query that filters on or returns any of the table's dimension foreign key fields must be filtered to a single dataset type, in fact, because only certain dimension columns are used for each dataset type.

This suggests that we should partition the dataset table into multiple per-dataset-type tables. These could each have only the dimension columns actually used for that dataset, and filtering on `dataset_type_name` would be implicit in the choice of table to query. These per-dataset-tyoe tables might need to be accompanied by a tall, thin `dataset` table that provides consistent autoincrement IDs across the per-dataset-type tables, along with a few genuinely common columns (which could be replicated in the per-dataset-type tables if helpful for performance reasons).

It is unclear to me whether this restructuring would improve the performance of typical queries on its own, though this seems at least plausible. In any case, it also adds flexibility that may be useful for other optimizations in the future. For example, it may turn out to be helpful for performance reasons to denormalize some of the columns in the visit (or observation) table into the table that holds the "raw" `dataset`. This is not really possible with a monolithic `dataset` table, but would be quite natural if "raw" had its own table.

The added flexibility of per-dataset-type tables is also useful for better data modeling. Per-dataset-type tables automatically provide a solution for the (minor) problem of where to put per-dataset-type metadata, for which we have long had a use case, but not a pressing need (and hence not an implementation). Combined with some kind of table indirection in query generation (see Generalized Observations and Multi-User Registries for other examples), this could also allow us to move dimension metadata into per-dataset-type tables where it may fit more naturally. For example, we could replace the `visit_detector_region` table with region columns in the dataset tables for e.g. "raw" and "calexp", and allow the Butler client to be configured to use either (though we would also need to make changes to how the spatial join tables are populated to make this work in full).

Perhaps most importantly, per-dataset-type tables make the system much more extensible, as they allow new dimensions to be added without altering a monolithic `dataset` table. That in turn allows new dimensions and the dataset types dimensions that utilize them to be developed in a user schema in a multi-user registry before being adopted in a shared schema.

The only drawback of this proposal that I currently see is that it means that registering a new dataset type involves a CREATE TABLE operation, not just an INSERT. That means it will be impossible for all butler tables (and indexes, etc.) to be created in advance by administrators, at least in user schemas. The DDL emitted when new dataset types are registered will still be generated by our Python code, however, and will almost always be quite similar to one of a few common dataset types whose DDL we will be able to optimize in advance.

~~ORM Lite/~~DDL Refactoring

The additional indirection and larger number of tables implied by many of the proposals on this page (e.g. multi-user registries, per-dataset-type tables, and generalized observations) will require a better separation between the interface and implementations of logical butler tables to keep the codebase complexity under control. At present we have a single `schema.yaml` file responsible for both, and tension between interface specification and implementation definition is already a blocker for features that require us to emit different DDL for Oracle and SQLite. At the same time, planned changes in the dimension system (**DM-17023** - Getting issue details... STATUS) will involve both caching records from individual [logical] dimension tables and adding Python code to the definition of some dimensions.

~~Together, these suggest a refactor that moves us closer to an Object-Relational Mapping architecture, in which we define the interface (i.e. field names and types) of each *logical* table via a `Record` class that represents the records of that table. These class objects would be passed to a separate `Backend` class responsible for emitting DDL (when necessary) and creating SQLAlchemy objects that correspond to the actual database entities. The `Backend` classes would be specialized for different databases, and would probably provide default implementations for arbitrary `Record` classes, as well as some specializations for important known tables. The `Backend` hierarchy may just be the `Registry` hierarchy itself, but it would also be worthwhile exploring a design in which `Registry` is concrete and final, more clearly separating the public interface of `Registry` from the conceptually protected implementation interface of `Backend`.~~

~~SQLAlchemy of course provides its own ORM layer, in addition to the lower-level database abstraction layer ("SQLAlchemy Core") we currently use exclusively. Using the SQLAlchemy ORM is of course worth considering, but I think we're better off writing our own and continuing to use only the core layer. That's in large part because what I believe what have in mind is vastly simpler than a full ORM, and I want the `Record` classes to be simple, lightweight, and immutable, with no automatic or implicit access to the database at all (probably implemented as `namedtuples` generated via a decorator similar to the new built in `dataclass`). But if we start down this path, we should be on guard against introducing unnecessary complexity (especially home-brew, not-invented-here complexity) in an attempt to add layering and encapsulation to manage our intrinsic complexity.~~

~~It should also be emphasized that this is not in anyway a proposal to hide or abstract away SQLAlchemy Core usage - we should absolutely continue to use SQLAlchemy objects to build queries, both inside the `Backend` classes and outside them.~~

After a small amount of quick prototyping, I think I'm ready to reject the ORM-lite idea, at least in the form described above. I *do* still think we should consider:

- making Registry concrete/final with a separate polymorphic Backend impleementation hierarchy;
- better separating the interface and implementation DDL for logical tables, by moving at least the latter fully to Python;
- defining at least *some* interface DDL (for dimensions) in Python (the rest could remain in schema.yaml).

But the extra metaprogramming complexity that would be involved in making that Python DDL specification a set of types whose instances can also be used as records just feels like a timesink with little upside. It'll be much easier to just hold the Python-side interface DDL specification in regular class instances (or even just nested dictionaries), and use SQLAlchemy constructs directly for the implementation DDL.

Configuration Tree Refactoring

Our configuration tree is currently organized according to the software components that consume configuration options, which is natural for developers and component extensibility. This organization combines configuration options that are set in very different ways at very different times, however, and this has complicated the development of the mechanisms that provide defaults and overrides for these options. We have at least the following categories of configuration options, from the perspective when they may be changed:

- Some configuration is essentially a declarative part of the codebase, and must be exactly in sync with the software versions of packages like `daf_butler`. These configuration values should probably always be retrieved from installed or EUPS-setup software locations.
- Some configuration is associated with a data repository, and is either used only when the repository is created or must be kept in sync with other content in the repository (e.g. database state), and hence should only be modified by special tools (if at all) after repository creation. Most registry configuration falls into this category.
- Some configuration is associated with a `Butler`, `Registry`, or `Datastore` client instance. A data repository or the codebase may provide defaults for these options, but unlike previous categories these options should be overrideable in class constructors and in the command-line arguments of scripts that construct these clients. Formatters and filename templates for a `PosixDatastore` are common examples of this category.
- Some configuration options are associated with both a particular data repository and a particular user. This includes authentication tokens and possibly default collections, and should generally be overridden *only* at the user level (or at least I can't identify a need for data repositories to provide a default for these options).

There may be yet more categories - a client+user category also seems likely, for example, though I can't think of any current options that would obviously benefit from being overridden in a user-space.

I think both the current component-based tree and something like the above "override categories" need to be explicit in the configuration tree. It's not obvious whether the tree is best organized first by component and then by override category, or the reverse. We could also consider treating the override category as a "tag" on individual options rather than a level in the hierarchy, but I think the configuration system must be able to generically inspect the override category of a particular option so it can decide whether to include it when writing config files in different contexts, and which source to give precedence to when merging configuration options from different sources.

Configuration Definition and Documentation

Our configuration currently doesn't provide a way for developers to document configuration options, or even declare what options are supported. Because the options supported at one level are themselves dependent on which component was selected at a higher level (i.e. which `Registry` or `Datastore` implementation), we can't even really use the common trick of providing a config file with all options specified and documented with comments (and defaulted options commented-out).

We should consider switching to a system for reading and validating configuration that lets us explicitly declare our configuration schema. That could actually involve using `pex_config` - possibly combined with some new functionality to allow `pex_config` to read and write YAML, which could be more generally useful as well. As messy as the guts of `pex_config` are, it does solve the problem of declaring and documenting configuration pretty well. We could also consider using a YAML declaration of the configuration *schema* (in the spirit of the JSON Schema project, or, I suppose, XML - perhaps something similar already exists for YAML elsehwere?), or perhaps something new analogous to `pex_config` in the sense of classes-as-configuration-declaration, but with YAML config files and `dataclass`-like class definitions.

Calibration Products Pipelines Challenges

One set of `PipelineTask` and data modeling use cases we have not yet prototyped thoroughly is in calibration products generation. There are already signs that those uses cases will involve functionality we don't yet support, and in some cases can't straightforwardly extend to support (at least not naturally).

The first problem is simply that the space of dimensions needed it label and relate calibration products generation is larger and qualitatively different from the current system of dimensions, which was developed primarily with on-sky data in mind and is hence centered around spatial relationships and standard observations. This has already proven to be a poor fit to laboratory test stand data, and the introduction of an AuxTel spectrograph and the Collimated Beam Projector as instruments will probably exacerbate this further. There is no generic solution for this; we (the middleware team) simply need to work carefully with the calibration products team to ensure we understand their data model and can map it to our system, modifying that system as necessary to accommodate it. Because this will almost certainly involve expanding the set of dimensions beyond the ~12 we have at present, it provides further motivation for partitioning and normalizing the dataset table, especially as this would also reduce the disruption involved in adding new dimensions later.

The second problem is that the current preflight system generally assumes that the relationships between datasets are defined fully by their dimensions, and that these dimensions (and their relationships) are defined prior to preflight, by non-`PipelineTask` "ingest" (raw, reference catalogs, bright object masks, master calibrations) or "registration" (skymaps) steps. This works well for on-sky data (including associating *master* calibrations with on sky data) because those relationships are trivially spatial or temporal - which `visits` overlap a `tract`, or which validity range encompasses an `exposure` (or `observation`). In much of calibration products generation, more of these relationships are user-input at execution time, and are much more subject to change; when making a master `flat`, for example, one of the inputs may be a (human-curated, git-controlled) file indicating which raw flat frames and which master bias to use as inputs for the master flat with a particular *provisional* validity range (which may change later after a complete set of master flats is produced). The natural way to incorporate that kind of input in the current system would be to load it immediately into one or more database tables (ideally temporary tables, at least in some cases) so we can use it in the kind of preflight queries we currently run, but it is worth considering whether `QuantumGraphs` for at least some calibration products pipelines should be generated via an entirely different algorithm.

Composite Datasets

We spent a significant amount of time early in the development of the Gen3 Butler designing an approach to composite datasets (e.g. exposures that contain PSFs), with the goal a system that:

- permits composite datasets to be be written monolitically with their components defined virtually;
- permits composite dataset to be defined as a virtual combination of a set of concrete component datasets;
- encapsulates this choice *per-dataset*.

In other words, an instance of the `calexp` dataset type could be written as multiple files in one run, but a single file in another, all within the same data repository, and a `Butler` accessing that repository would be able to query for and access those datasets in exactly the same way.

This flexibility comes with a cost (at least with the current design): each component of a composite dataset always gets its own entry in the `dataset` table, even though most of these components will never be accessed independently in practice. These little-used components currently represent a little more than half of all of the records in the `dataset` table (in the `ci_hsc` data repository, at least), and the ratio will probably get worse in the future as we define new components.

We also have not yet actually realized all of the benefits of this system, and it is unclear if we actually need them. The preflight algorithm currently does not permit component datasets to be used as inputs, as it does not recognize those as dependencies satisfied by the creation of the parent composite dataset by some other `PipelineTask`. We have never even worked out at a conceptual level an approach that would permit a virtual composite to be defined from separately-produced components within a Pipeline (such as allowing coaddition to declare a dependency on a calibrated exposure comprised of an image produced by one task in the pipeline and a WCS produced by another).

I still worry that revisiting the current design would be a time-sink with little gain at the end; what we have now does probably work well enough for the use cases we have right now, and may work well enough going forward. But the design has probably been contorted (in a way that may affect performance and maintainability) by nice-to-haves that we've not actually ended up supporting, and it would be nice to have the time to re-evaluate what functionality we really need and whether a simpler (and more efficient) approach could support that.

Multi-Collection Butler and Collection Chaining

For the most part, a Gen3 data repository maps to multiple Gen2 data repositories, with the closest Gen3 analog to a single Gen2 repository being a collection.

In Gen2, however, data repositories can be chained, so a dataset can be "in" a repository by virtue of being (directly) in one of its parents. That kind of inherited repository membership is in many respects different from direct membership in a repository. It can be changed, for example, by the addition of a new dataset in the child repository that shadows that in the parent. It is also *lazy* - the set of datasets in a Gen2 repository cannot be iterated directly, so membership can only be tested by an explicit test on a particular data ID and dataset type. This hides some confusing aspects of this kind of membership; if one creates a child repository containing the outputs from processing a small number of raw images, it might be surprising to find that all of the raw files in the parent repository are considered part of the child.

Making data repository (and collection) contents iterable is a major goal (and one of the most frequently requested features) of the Gen3 Butler, and as part of that we have adopted a design for collection membership that does not have any notion of implicit membership or chaining. A dataset is present in a collection if and only if there is a record in the `dataset_collection` table for that dataset and collection. To simulate the parent-repository chains generated by Gen2 processing runs, we have instead planned to *explicitly* add datasets from the input collection to the output collection, but this requires a subtle choice on what to include:

- Should we add *all* datasets from the input collection to the output collection, regardless of whether they are used in the processing run, as long as they are not shadowed by a new dataset? This preserves the Gen2 behavior exactly, but (as noted above), that behavior might be considered surprising when collection contents are iterable, and it certainly adds many records to `dataset_collection` that will never be used.

- Should we add only those datasets that are used directly as inputs by the tasks being run? This might confuse users accustomed to the Gen2 behavior, particularly because it makes the (simulated) chaining non-transitive: if one produces `calexps` from `raws` in one run, and then coadds from those `calexps` in another, the coadd collection will not contain raws. This choice minimizes the number of probably-unused `dataset_collection` records, of course.

- Should we add datasets that are used as inputs, but utilize the provenance of those datasets to recursively add their inputs? This makes collection chaining transitive, without what I'd consider a confusing degree of expansion, but it still adds a lot of probably-unused `dataset_collection` records, and it's still a change from the Gen2 behavior. And while the provenance necessary to do this is in the database (well, could be soon if we prioritize that), it may still be expensive to recursively query for it.

I'd argue that the best behavior, in terms of both minimizing the size of `dataset_collection` and unsurprising behavior, is actually none of these. Instead,

- We should not add input datasets to output collections at all (except perhaps as a rarely-used alternative option).
- We should allow a Butler to be initialized with a collection search path (like the one the preflight algorithm already accepts) instead of a single collection.
- We should associate (in the database) the input search path used when creating an output collection, and provide an easy way to construct a Butler with a search path defined by a collection and the search path used to create it.

Together, these changes would make collection iteration never yield "inherited" datasets, which is probably the least surprising behavior, while maintaining the ability to obtain transitive inputs via an explicit `Butler.get` call. Changing Butler to use to use a collection search path is moderately straightforward; the `SingleDatasetQueryBuilder` class used in preflight already implements the query needed to search the dataset according to a fairly complex specification of ordered and dataset-type-specific collections. Storing that search path in the database probably necessitates adding a new `collection` table, which might spell the end of implicitly-created-when-needed collections. But doing so also creates an opportunity to use a more compact autoincrement integer primary key to identify collections in other tables (particularly `dataset_collection`), and it may help guard against accidental collection creation via typo as well. The changes to the Butler construction API are easy implement but merit careful thought in terms of the desired user interface; we want to make it easy to load a collection's default search path while still making it straightforward to ignore or override it.

My biggest concern with this proposal is that it implies a provenance relationship where one only *usually* exists, though this is a concern with any approach that mimics the Gen2 behavior. The Gen3 butler tracks provenance at a per-dataset level, instead of a per-repo or collection level, and hence it has the necessary information to answer much more precisely questions like "which calexps were used to generate this coadd", which is often what a user is really asking when retrieving a dataset from a parent repository. In fact, using the collection search path naively in Gen3 is actually slightly more likely to be misleading about provenance than parent lookup was in Gen2, because Gen2's support for multiple parent repositories never made it into the `CmdLineTask` driver and hence was extremely rarely used (and conflicts between parent collections/repos is the primary way collection/repo provenance can differ from per-dataset provenance). That said, such confusion about provenance should still be quite rare, and is probably best addressed by making sure we provide intuitive (and well-documented) APIs for asking provenance questions properly.

Immutable DatasetTypes, DataIds, and DatasetRefs

The objects we used to describe datasets can all exist in several states, generally in a sort of progression from "unvalidated/incomplete" to "validated /complete":

- DatasetTypes may have their dimensions defined by a DimensionNameSet (essentially just a set of strings), or a DimensionGraph, which is guaranteed to only contain valid dimensions, self-consistented expanded to include dependencies, and topologically ordered.
- DatasetTypes may have only the string name of a StorageClass, or a real StorageClass instance.
- Data IDs always have a full DimensionGraph, but may have any amount of dimension metadata in their entries attribute (some of which may be needed to create DataIdPackers, and fill in filename templates later).
- DatasetRefs may or may not have a dataset_id, which both indicates known existence within the Registry and association with a particular set of Collections.

The fact that the state of these objects can change at all makes caching them to avoid database queries (which is very important for performance) a bit dangerous. But the fact that the *kind* of state they can have is much worse - it makes it impossible to define unsurprising comparison operators for them, and these are all objects that are used as keys in dictionaries and sets. In particular, a DatasetType with unexpanded dimensions can never be compared to each other, and they can only be compared to a DatasetType with expanded dimensions if we do the expansion during the comparison, utilizing the expanded operand's pointer to the dimension universe. That implies that comparisons should implicitly modify the unexpanded operand by expanding it (surprising behavior!) or compare something other than operand (also surprising!). And our current implementation doesn't do either of these - it just blindly compares the dimension lists, yielding occasionally-incorrect comparisons. The situation is both more straightforward and harder to resolve for DatasetRef, which could be compared on either its dataset_id or the combination of its DatasetType and DataId. These different questions of course yield different answers, and users in different contexts will expect one and be confused by the other. Our current implementation for DatasetType also does neither, instead blindly comparing all attributes, including many that should not be considered comparison keys.

I think the best way out of this mess it to simply drop the unvalidated/incomplete versions of these classes, and at the same time make them immutable. I think we already have other objects we can use in contexts where it's impossible to validate or complete the objects (generally because the objects haven't been rendezvoused with a Registry). In detail:

- DatasetType should *always* hold a DimensionGraph (StorageClass is less important, because that completion is already invisible to the user, but I think we can simplify the code by saying it always holds a StorageClass instead of just the name of one, too). The main use case for unvalidated /incomplete DatasetTypes is in the PipelineTask methods that report the task's input and output DatasetTypes. But those already use custom classes, because those methods also report things like whether the task expects one or many instances of that DatasetType per Quantum, and we could easily modify those classes to hold the dataset type name, set of dimension names, and storage class name directly instead of holding a incomplete DatasetType object.
- DataId objects should always be fully expanded to include all metadata that *could* be used to complete templates or construct DataIdPackers, and any other metadata in the database should *never* be held by a DataId (as opposed to, say, being in a dict indexed by a DataId). The exact boundary of what should or should not be included is not yet clear, and it's possible that we'll waste time always loading more metadata than we'll actually use. Right now, however, we're actually spending more time trying to identify what metadata we might need to query for relative to what a DataId already has, and by always making DataId objects maximimally expanded, we'll make it much more straightforward to cache them and avoid most database lookups. Plain Python dicts are already accepted by most high-level Butler APIs and can continue to serve as an object that represents an incomplete/unvalidated data ID.
- DatasetRefs should always represent a known-to-exist (at least in a Registry) dataset with a valid dataset_id. While we don't have a single object we can already use to represent a possible future dataset, we can already just use the combination of a DatasetType and a DataId for that purpose (as we frequently already do). The only major context where we use DatasetRefs to indicate possible future datasets is in the Quantum objects generated by preflight, but there they're already in dictionaries keyed by DatasetType, so we can just switch to DataIds with no loss of information - a few loops that iterate over the .values() in a Quantum's dictionaries would just have to iterate over .items() instead.