Architectural Prototype for the New Gen3 Registry

Goals

- Provide a vision for how a `Registry` implementation that uses different database schemas/namespaces (in the same database) for different users could be written, while deferring most of the work on that task for the future (instead assuming a "friendly-user / shared schema" model for the time being).
- Reduce usage of SAVEPOINTs and (SQLite) exclusive locking while maintaining concurrent transaction safety.
- Adding vectorized dataset ingest (which involves obtaining autoincrement primary key values after insertion, and the interaction between that and concurrent transactions and multi-layer Registries).
- Adding vectorized collection-association (which involves replacing when UNIQUE constraints are violated, and the interaction between that and concurrent transactions and multi-layer Registries).
- Adding an "origin" field to dataset (and quantum) to give it a primary key that is more than just autoincrement.
- Using integer surrogate IDs in primary and foreign keys even for entities uniquely identified by strings, such as dataset types and collections.
- Supporting dynamic (on-demand) table creation, which will in general interrupt transactions and thus has to be treated with care.
- Adding flexibility to the constraints on data ID + dataset type uniqueness within collections (desired for some combinations of datasets and collections, but not all; see Dataset/Collection Constraints below).
- Moving the validity ranges for calibration datasets from the dimensions system to the association between a dataset and its collection, so the same dataset can have a different validity range in different calibration collections.
- Better normalize the dimension columns in the dataset table(s) and add dimension columns to the quantum table(s), while abstracting over and encapsulating the specific DDL possibilities that could be used to achieve this (see e.g. Dataset and Collection Table Reorganization).
- Improve `Registry` maintainability, even while adding all of the above sophistication. This is mostly about improving separation of concerns; `Registry` is pretty much a classic god class right now. In particular, we need to separate various schema design concerns both from each other and from the concern of adapting to the particulars of specific database engines.

I believe the architecture described below achieves all of these goals (even if it is sometimes vague about the details). It also addresses (in part) some issues I didn't set out to work on at all:

- It provides a vision for how to define a chained Registry across different database engines and backends. Such a Registry would not have all of the functionality of a single-database Registry, but the architecture makes it clear which operations can always be supported and which operations would be limited
- It helps with schema migration problems by allowing different layers in a multi-layer Registry to have different schemas.
- It reduces our unit testing surface, by making it possible to get coverage on tests for different database engines over a much smaller API.

Overview and Glossary

The new **`Registry`** is a concrete class that defines the only interface (of those described here) that is considered public to code outside `daf_butler`. It is a thin wrapper around `RegistryLayer` that performs error-checking, friendly error messages, and type coercion before delegating any real work to the classes below. In the near term, `Registry` will have a single `RegistryLayer`; in a multi-layer `Registry` future, it will have an ordered list of `RegistryLayer` instances, and delegate to them in turn.

`RegistryLayer` is a simple struct-like class that aggregates a `Database` instance and several "manager" classes responsible for different kinds of entities that can exist in a `Registry`.

`Database` is an abstract base class whose instances represent a particular schema/namespace and provide low-level database operations. `Database` subclasses correspond to different database engines (SQLite, Oracle, etc).

The manager classes are all abstract base classes that act as containers and factories for an associated "record" or "records" class (also an ABC, unless otherwise noted below):

- **`CollectionManager`** manages the database representation of collections and runs within a single layer, often via **`CollectionRecord`** (a simple concrete struct) and **`RunRecord`** (an ABC that allows limited updates to runs) instances.
- **`OpaqueTableManager`** manages the opaque tables used to store `Datastore` internal records. It acts as a container and factory for **`OpaqueTableRecords`**, which represents a single named table (which usually maps to a single `Datastore` instance).
- **`DimensionTableManager`** manages the dimension element tables and their join tables. It acts as a container and factory for **`DimensionTableRecords`**, which represents the table(s) for a single dimension element.
- **`DatasetTableManager`** manages the dataset tables and the tables that relate them to collections. It acts as a container and factory for **`DatasetTableRecords`**, which represents the table(s) for a single dataset type.
- **`QuantumTableManager`** manages the quantum tables and the tables that relate them to datasets. It acts as a container and factory for **`QuantumTableRecords`**, which represents the table(s) for a particular set of dimensions.

All manager and records classes have a reference to the `Database` that backs their layer. `Registry` interacts directly with the manager and records objects, but rarely (if at all, at least after construction) with `Database`.

The prototype itself is on branch u/jbosch/prototyping, and mostly in the daf.butler.prototyping subpackage (more direct pointers to useful content appear later in this document). None of this code can be expect or even import cleanly, but the documentation for the interface classes is ready for review. The implementation classes are best considered an in-depth thought experiment to check that the interfaces were viable.

Prototype Details

Changes to Concepts and Primitives

The current codebase has classes for `Run` and `Quantum` that subclass `Execution` (which is not used otherwise), while collections are always represented as strings. The relationship between a run and its collection is ambiguous and ultimately unsound - it provides a loophole that blocks us from being able to guarantee unique filenames (at least). The prototype drops the `Execution` class (and table), and instead treats a run as a kind of collection (see Dataset/Collection Constraints). Both runs and collections are now represented by strings in high-level code, and by `CollectionRecord` and `RunRecord` (which provide access to the integer surrogate ID) within `Registry` and its component classes. The host+timespan information provided by `Execution` has now just been moved directly into the run and quantum tables.

Registering collections (not just runs) is now explicit, not something that happens implicitly when a collection is first used.

The prototype codebase currently uses new `DatasetHandle` classes instead of `DatasetRef`; `DatasetHandle` has different classes that represent different levels of knowledge about a dataset (in particular, "resolved" means we know its ID and origin, and that it exists). I'm not sure, in retrospect, that the split into multiple classes was worthwhile, and it may be reverted, but this is largely tangential to the rest of the architecture, and it is *not* worth reviewers' time to think about.

The prototype codebase also uses special classes to represent iterables of `DatasetHandle` and `DataCoordinate`. These were motivated in large part by a desire to have containers/iterables that remember what "optional" aspects their elements have - in particular, whether datasets are resolved and data ID are expanded. The existing prototype code doesn't really utilize this much, and they may not be a part of the final design. I'd like to see how they interact with the high-level Registry interface and particularly its query operations before deciding; they could also be a very nice way to represent query results.

Dataset/Collection Constraints

The prototype defines two new enumerations that together describe the kinds of constraints we place on the datasets that can be in a collection.

First, there are three types of collections:

https://github.com/lsst/daf_butler/blob/u/jbosch/prototyping/python/lsst/daf/butler/prototyping/interfaces/collections.py#L21

```
class CollectionType(enum.IntEnum):
    """Enumeration used to label different types of collections.
    """

    RUN = 1
    """A ``RUN`` collection (also just called a 'run') is the initial
    collection a dataset is inserted into and the only one it can never be
    removed from.

    Within a particular run, there may only be one dataset with a particular
    dataset type and data ID, regardless of the `DatasetUniqueness` for that
    dataset type.
    """

    TAGGED = 2
    """Datasets can be associated with and removed from ``TAGGED`` collections
    arbitrarily.

    For `DatasetUniqueness.STANDARD` dataset types, there may be at most one
    dataset with a particular type and data ID in a ``TAGGED`` collection.
    There is no constraint on the number of `DatasetUniqueness.NONSINGULAR`
    datasets.
    """

    CALIBRATION = 3
    """Each dataset in a ``CALIBRATION`` collection is associated with a
    validity range (which may be different for the same dataset in different
    collections).

    There is no constraint on the number of datasets of any type or data ID
    in ``CALIBRATION`` collections, regardless of their `DatasetUniqueness`
    value.

    There is no database-level constraint on non-overlapping validity ranges,
    but some operations nevertheless assume that higher-level code that
    populates them ensures that there is at most one dataset with a particular
    dataset type and data ID that is valid at any particular time.
    """
```

and (in this respect) two different kinds of dataset types:

https://github.com/lsst/daf_butler/blob/u/jbosch/prototyping/python/lsst/daf/butler/core/datasets/type.py#L40

```
class DatasetUniqueness(enum.IntEnum):
    """Enumeration indicating the kind of uniqueness constraints to
    define on data IDs of this dataset type.
    """

    STANDARD = 2
    """There may only be one dataset with this dataset type and a particular
    data ID in a `~CollectionType.TAGGED` collection.
    """

    NONSINGULAR = 3
    """There may be any number of datasets with this dataset type and a
    particular data ID in a `~CollectionType.TAGGED` collection.
    """
```

This table summarizes the constraints we place, given both of these:

	DatasetUniqueness.STANDARD	**DatasetUniqueness.NONSINGULAR**
CollectionType.RUN	data ID + dataset type	data ID + dataset type
CollectionType.TAGGED	data ID + dataset type	no constraint
CollectionType.CALIBRATION	no constraint*	no constraint

* there are no constraints on CALIBRATION collections in the database, but we expect the higher-level code that produces them and assigns validity ranges to ensure that there is no duplication within each validity range when that makes sense for the dataset type.

Implications for Calibration Products Production

The introduction of CALIBRATION collections with validity ranges is tied to the removal of validity ranges from the dimensions system, where they are currently linked to the `calibration_label` dimension. Whether the `calibration_label` dimension itself is removed as well depends on how we want to use RUN collections when building master calibrations.

Once a validity range is known for a master calibration, it can be associated with a calibration collection, and that validity range added to that association can be used along with the rest of its data ID (e.g. `instrument+detector` for bias, `instrument+detector+physical_filter` for flats) to retrieve it any given point in time. I am relatively confident that this a solid model for *using* master calibration datasets, at least once a complete suite of calibrations is produced.

Before it is associated into a calibration collection, however, we still need a way to uniquely identify a master calibration, at the very least so we can later identify which dataset we're associating with each validity range. In Gen2, we used a `calibDate` field for this, which no one was happy with, and that's what the `calibration_label` was intended to replace (with the *content* of that label still very much TBD; it could be date based, or computed from the constituent visits, or a counter...): the full data ID for a bias would be `instrument+detector+calibration_label`. With `calibration_label` no longer needed for validity ranges, we now have two ways to proceed:

- We could continue to use `calibration_label` as part of the data IDs for master calibrations (and possibly rename it to reflect how we expect to populate it, once that's decided). This would allow us to have multiple calibrations of the same type and "rest of the data ID" in the same run (i.e. multiple master biases for the same `instrument+detector`, which would *eventually* be mapped to different validity ranges), which would be most natural if we expected to produce them at once (i.e. in a single `QuantumGraph` execution). But we would have to add logic to the lookup within calibration collections to not require the `calibration_label` part of the data ID there, because it would be redundant with the validity range lookup.
- We could drop `calibration_label` entirely, and put master calibrations that would otherwise conflict in different runs as they are created, so (e.g.) within any run an `instrument+detector` lookup for bias would be fully qualified. This would be the most natural approach if we really do only want to produce master calibrations one-set-at-a-time with explicit validation before blessing them (which is here expressed as assocation with calibration collection), and it saves us from having to cook up something to put in the `calibration_label` string. But as all other `QuantumGraph` execution puts all outputs datasets into a single run, we would either need to add more special-casing or levels of indirection to that higher-level code or rule out any single-`QuantumGraph` method of producing conflicting master calibrations. (To be clear, this would be a prohibition on e.g producing multiple biases for the same detector in a single QG; there would not be any prohibition on producing a single master bias for each detector in one QG).

The `Database` Class and SQLAlchemy Usage

`Database` fully encapsulates the SQLAlchemy connection and engine objects, requiring all write operations to go through specialized methods. That works because we seem to only need relatively simple version of these operations, and in at least some cases it's necessary because we depend on non-standard functionality that most database engines nevertheless support, but in different ways (e.g. sync, replace). Encapsulating write operations also allows `Database` complete control over transactions, including the handling operations that can interrupt transactions (both table creation and sync, which may require retries inside its own transaction to be safely concurrent for some database engines). We consider performing any operation that could interrupt a transaction in within a transaction block a logic error, regardless of whether it actually would in the database in question. Happily, all of our high-level operations that need to be implemented on top of transaction-breaking low-level operations (e.g. registering new dataset types, adding new collection) seem to be the kind where this is not a major problem.

`Database` does not attempt to provide much of an abstraction around SQLAlchemy's tools for constructing and executing SELECT queries. Fully encapsulating SQLAlchemy for SELECT queries would be a huge waste of our time, as we'd essentially be reimplementing a large chunk of SQLAlchemy's own abstracts. And we don't have to – while we will be writing fairly complex queries, we don't expect to need to use extensions to standard SQL.

Table definition and creation follows a middle path. We have our own classes (not just in the prototype, but on master) for representing table, field, and foreign key specifications. The manager and records objects use these to describe what they need to a `Database`, which can then customize the translation to SQLAlchemy before returning the SQLAlchemy representation to the manager and records objects. A major change from the current `Registry` implementation is that the schema is now defined programmatically by the manager and records classes, instead of in YAML configuration. This keeps them close to the code that depends on them, and it allows us to encapsulate normalization and other schema design changes behind stable Python interfaces.

The result of these choices is that `sqlalchemy.sql.FromClause` and `sqlalchemy.schema.Table` objects (and the objects that comprise them) are quite common at the manager and records interface level. Other SQLAlchemy objects are mostly hidden behind `Database`, and no SQLAlchemy usage is exposed in the public `Registry` interface.

The prototype for the `Database` interface is nominally complete, in that no further changes or additions seem necessary right now (aside perhaps from clearer expectations on exceptions that will be raised for different error conditions). Small changes as it transitions from a prototype to production code are of course expected. There is no concrete implementation of `Database` in the prototype, but the base class does provide default implementations for most methods (and will provide more in the production version). The ABC does have complete API documentation, and that is worth reading for reviewers interested in more detail.

Manager and Records Classes

The manager and records interfaces are structured largely to encapsulate decisions about to map our in-memory data structures to tables, including both normalization/denormalization and vertical/horizontal partitioning. This encapsulation solves an number of problems:

- Adding new implementations of the same interfaces without removing the old ones would let us continue to support old repositories after a schema migration without changing anything on-disk or in-database.
- Different aspects of the schema handled by different interfaces, so we can experiment with different combinations of choices without needing a combinatorial number of code branches.
- When we have multiple layers, different layers will be able to use different manager and record subclasses within the same `Registry` .

In some cases, these interfaces also provide an abstraction over whether entities that are relatively few in number (collections, dataset types) are aggressively fetched from the database to avoid repeat queries, fetched only when needed, or something in between (i.e. cached after first use).

The separation into managers and records classes establishes a boundary between operations that depend only on "static" tables that can be expected to always be present in any layer (all a manager class interacts with) and those that may also depend on a "dynamic" table (one that is only created in a layer when needed). Table creation can be transaction-interrupting, and the same is actually true of the `Database.sync` operation we would use on the static "meta" tables that remember which dynamic tables exist, so it's very important that the API make it clear when table creation could happen. The two-class pattern achieves this by putting all transaction-interrupting operations inside the manager class's `register` method, which is one of two ways a records object can be obtained (and the other, a `get` method on the manager, simply returns `None` if no such records object already exists). The `CollectionManager` is a partial exception to this pattern - we do not expect to need any dynamic tables for collections, but we do need to use `Database.sync` when registering a new collection.

The manager and records interfaces do not attempt to encapsulate decisions on what kinds of primary keys we would use for different entities. The interfaces explicitly require each entity to have one of three distinct kinds of primary key; it turns out that the kind of primary key has implications even for the primitives that represent this entities in Python, so abstracting over those choices does not seem worth the effort. In particular:

- Datasets and quanta are identified by autoincrement integer IDs, and because these are the only unique identifiers for these (or at least the only simple ones), we use compound primary keys that include another "origin" integer that allows them to be transferred between repositories or layers without rewriting IDs (which assumes that we have some way of assigning unique origin values across potentially-related repositories).
- Collections (including runs) and dataset types are identified by string names that we require users to provide, but we internally (and only internally, never in public Registry APIs) use a surrogate autoincrement integer primary key for these. As these IDs are completely internal, we never try to maintain them when transferring between data repositories or layers, and there's no need for a two-component ID that includes an origin value.
- Dimensions have integer or string identifiers that are provided by users or higher-level code, and we use those directly as primary keys (essentially demanding that higher-level code guarantee their uniqueness).

Another common pattern in the manager and records interfaces is the presence of one (or occasionally two) `select` methods that return `sqlalchemy.sql.FromClause` objects. Those may represent tables directly if appropriate, but more frequently will represent SELECT queries that can be used as subqueries to form more sophisticated queries. This approach has already been worked out to a large extent on master in the queries subpackage, and while that will need changes to work with the new prototype, I'm not concerned about the viability of the approach overall. The `select` methods (and the higher-level query APIs that utilize them) are significant for the extension to multi-layer registries, however, because they represent the only operations for which a straightforward "try layers on order until one succeeds" (or an equally simple variant on that) doesn't work. Instead, the query system for a multi-layer `Registry` would call `select` methods on all layers, drop those for which the result is `None` (which may happen if, e.g., the layer knows it has no datasets of a particular dataset type), and combines the rest via UNION or UNION ALL before passing the result of that to the query builder system (which could otherwise remain largely unchanged). That means these operations are the only operations that cannot work in general on a `Registry` backed by layers with heterogeneous database engines, and even there they could sometimes work if the logic is clever enough to notice when all layers that actually contribute to a particular query are backed by the same connection.

The manager and records interfaces sometimes depend on each other:

- `DatasetTableManager/Records` depends on `CollectionManager/CollectionRecord/RunRecord` instances to both translate names to/from their internal integer IDs, and to add foreign key fields to its own tables that reference the collection and run tables.
- `DatasetTableManager/Records` *construction* depends on the `QuantumTableManager/Records` subclass (but does not require an instance) in order to add foreign key fields that reference the quantum tables.
- `QuantumTableManager/Records` depends on both `CollectionManager/CollectionRecord/RunRecord` and `DatasetTableManager/Records` instances for both foreign key fields and mapping database IDs to Python representations.

The abstract base classes for all of the manager/records pairs are mostly complete and fully documented in the prototype, and are worth looking at for further details:

- `CollectionManager/CollectionRecord/RunRecord`
- `OpaqueTableManager/Records`
- `DimensionTableManager/Records`
- `DatasetTableManager/Records`
- `QuantumTableManager/Records`

There are probably some operations missing (querying collections and handling composite datasets come to mind), but enough is present to give me confidence that the overall architecture is stable. I believe all operations that involve nontrivial interactions between different components have been included.

The prototype also includes reference implementations for all of these, but these are undocumented and completely untested; as noted above, they were written primarily as thought experiments to validate the interfaces (though I do intend to copy from them heavily when writing the production versions). I'm not sure we will ever need more than one implementation for most of these (collections and opaque tables in particular are pretty straightforward), but it nevertheless seems prudent to separate interface from implementation for all of them.

Registry Itself

While there is a `Registry` class in the prototyping directory, it's not nearly as complete as the rest of the prototype I've described here - not only is it interface-only right now (despite the vision of it being a concrete class), but I haven't actually looked at it at all since the rest of the prototype came together in its current form. I do expect there to be changes to the public `Registry` API associated with moving to the prototype, but these should be straightforward: separating transactional from transaction-breaking operations, adding vectorization, and using more consistent verbs for similar operations on different entities.

I'm of two minds about whether to continue the prototyping work to extend it to `Registry` itself, or to just dive into moving the existing prototype ideas into the production version and updating the Registry APIs in the process. I don't think I need to treat the implementation of `Registry` itself as another thought experiment to establish the viability of the overall architecture of the prototype, but I do imagine that doing so would help clarify and stabilize the details of the manager/records and Database APIs before those go into active use. This needs to be weighed against the need to start actually putting a lot of this functionality onto master sooner to unblock others (especially the vectorization and SAVEPOINT reduction that will speed up at-scale ingest to the point where it's actually usable).

Remaining Challenges for Multi-Layer Registry

In a single-layer Registry, all of our logical foreign key constraints and unique constraints can be real database constraints. This is not true in a multi-layer Registry:

- Logical foreign key constraints would need to reference the union of several real tables, which simply isn't possible in most (any?) database engines. This can be mitigated in some cases by copying database entities into a layer when they are used in that layer; this seems like the right approach for collections and dataset types, which will always small numbers of (relative to datasets and quanta, and at least some dimensions). It is certainly the wrong approach for other entities - most user-level registries will never need to include *any* dimension tables, as they'll be able to exclusively use those defined in a shared base layer. For these we need to evaluate how hard it is to impose these constraints in higher-level code, and how problematic it is if we can't do so rigorously or concurrently.
- Logical unique constraints would either need to be applied to the union of several real tables, or we would need to resolve duplicates in favor of higher-priority layers when combining them in queries. The former would again involve imposing the constraint outside the database, and here that definitely seems difficult while also being important to do rigorously (particularly for dataset/collection constraints). On the other hand, trying to resolve duplicates in query results seems like an expensive, always-on solution to a rare-in-practice problem. Some way of carefully guaranteeing less about uniqueness or duplication in query results that still meets higher-level use cases seems the better way to go, if we can thread that needle.

The other obvious gap in this design in moving to multiple layers is in Registry construction and configuration. I think this would need to involve at some level specifying concrete class names for the managers/records and Database in configuration, but there are simply a lot of details to work out. It does seem to be a much simpler problem if we prohibit "rebasing" of layers - once some layer B has been created on top of some other layer A, one would never be able to load B without A. Hopefully we will be able to maintain flexibility in the opposite direction (not requiring B in order to load A), of course.