LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)

The Gen3 Butler Registry Schema

Jim Bosch

DMTN-073

Latest Revision: 2018-05-19

Abstract

Documentation for the SQL schema that will be used to manage datasets in the Gen3 Butler.

Change Record

Version	Date	Description	Owner name
	2018-02-19	Initial version.	J. Bosch
	2018-03-27	Ready for dev team review.	J. Bosch
	2018-05-19	Ready for DM-wide RFC review.	J. Bosch

Document source location: `https://github.com/lsst-dm/dmtn-073`

Contents

The Gen3 Butler Registry Schema

1 Overview

This document is a human-readable description of the minimal SQL schema that will be used in the Gen3 Butler's Registry component.

While some Registry instances may have additional tables, all must provide at least the tables and views described here, and are generally expected to use the mechanisms described here for most extensions.

The normative, machine-readable version of the minimal schema can be found at:

`daf_butler:config/schema.yaml`.

Most of the tables and figures in this document (including the descriptions of table columns) are generated from the contents of that file.

The current SQL schema should be considered tentative and conceptual; we expect a round of normalization/denormalization changes to be driven by performance concerns in the future. In order to reduce future disruption from such changes, we'd like to identify and fix now any aspects of the schema that are both guaranteed to cause performance problems and have obvious solutions, but we would like to avoid hypothetical optimization discussions until we have an opportunity to see how the schema performs under realistic conditions.

2 Datasets

2.1 Dataset

The Dataset contains a single record for every discrete unit of data managed by the Registry, and acts as a sort of hub for the rest of the schema: nearly all other tables join to it, either to label datasets (Section 3), provide provenance information and define groups (Section 4), or connect to the Datastores that actually store them (Section 5).

Finding a particular dataset (assuming one does not already have the primary key value or provenance information) typically requires three pieces of information:

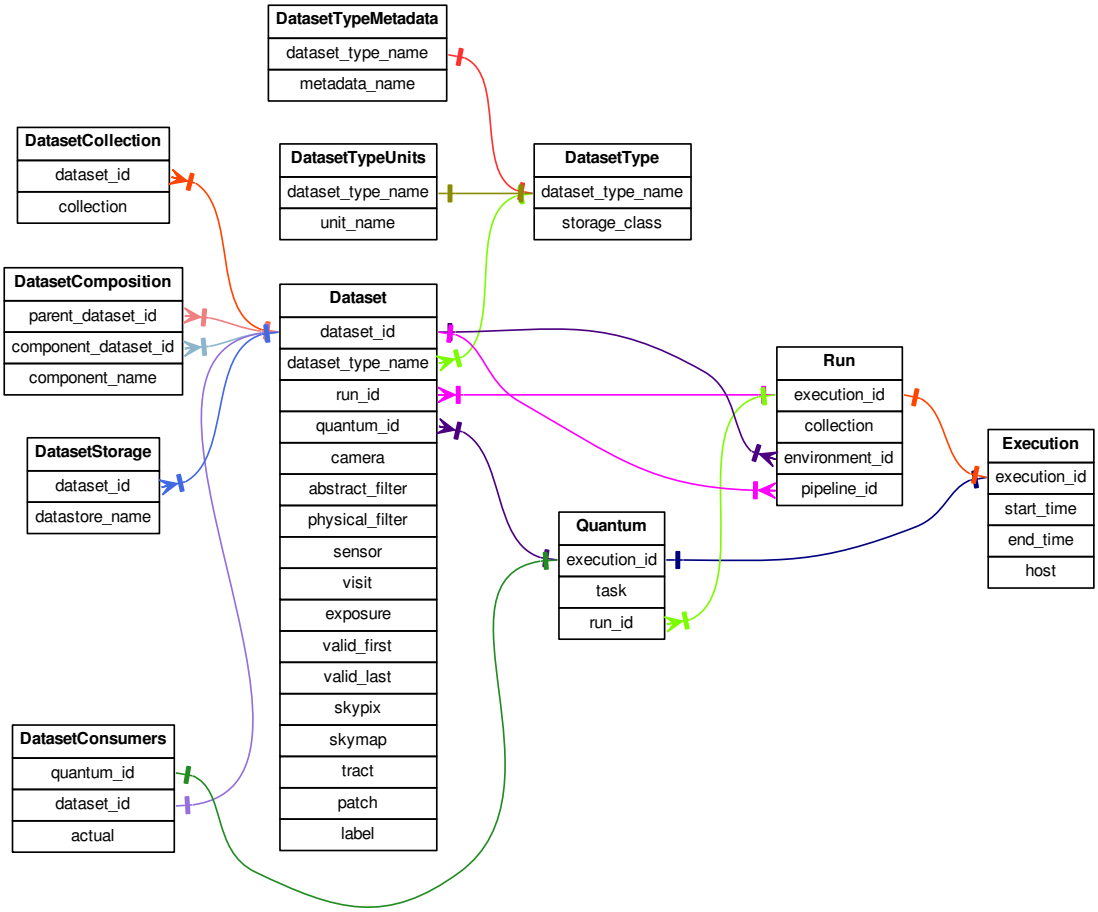

FIGURE 1: Table Relationships for the *limited* schema, which lacks DataUnit dimension and join tables. Colors are for disambiguation only.

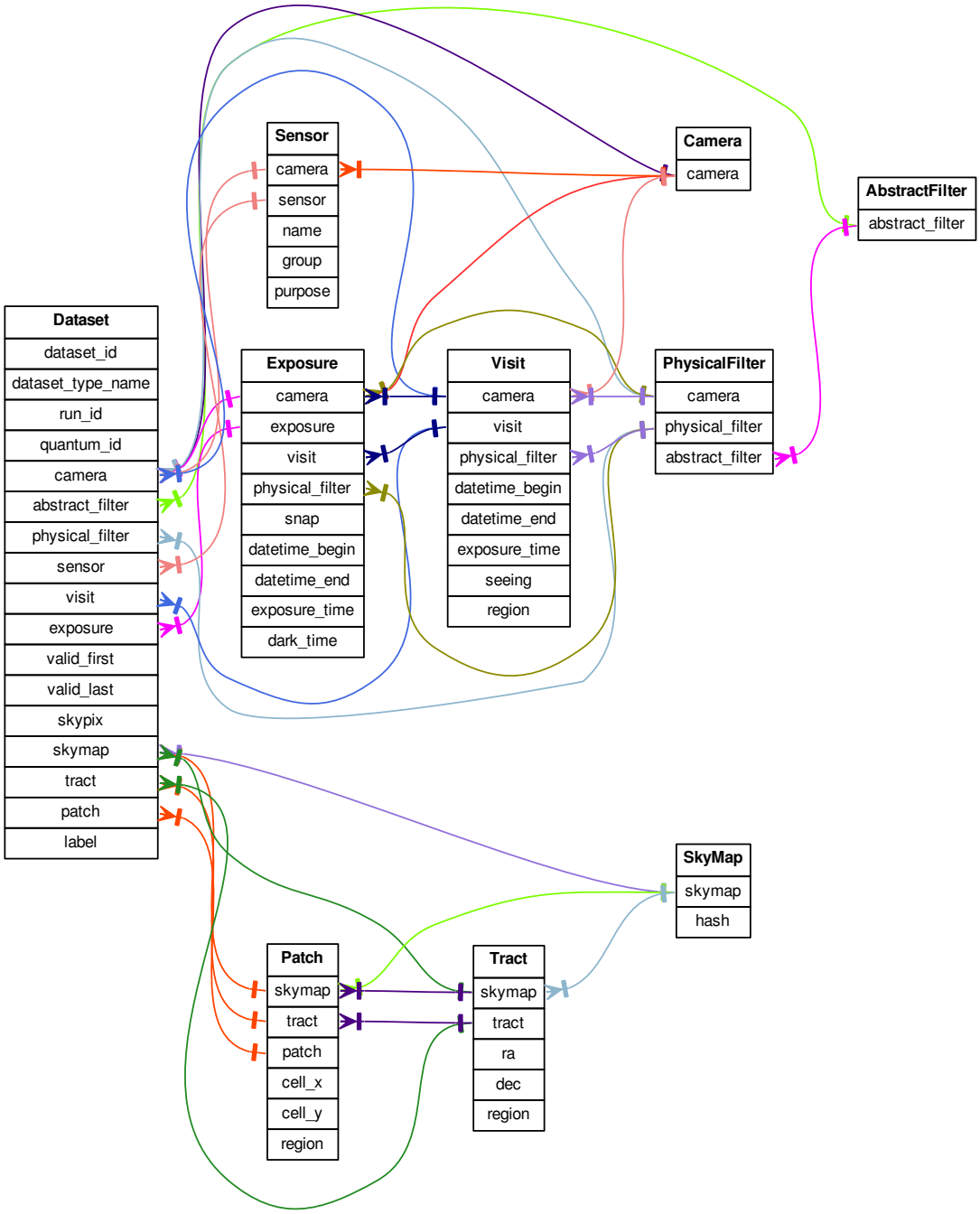

FIGURE 2: Table Relationships, DataUnits and Dataset. Colors are for disambiguation only.

- its DatasetType (e.g. `src`);
- one or more Collections to search;
- a set of DataUnit values.

The full set of Dataset columns can be found in Table 1.

Name	Type	Attributes	Description
dataset_id	int	PRIMARY KEY	A unique autoincrement field used the primary key for dataset.
dataset_type_name	str	NOT NULL	The name of the DatasetType associated with this dataset; a reference to the DatasetType table.
run_id	int	NOT NULL	The id of the run that produced this dataset, providing access to coarse provenance information.
quantum_id	int		The id of the quantum that produced this dataset, providing access to fine-grained provenance information. may be null for datasets not produced by running a supertask.
camera	str		
abstract_filter	str		String name for the abstract filter, frequently a single character.
physical_filter	str		
sensor	str		
visit	int		
exposure	int		
valid_first	int		First exposure identifier included in the range (inclusive). may be zero to indicate an open interval.
valid_last	int		Last exposure identifier included in the range (inclusive). may be max(int) to indicate an open interval.
skypix	int		Unique id of a pixel in the hierarchical pixelization, using a numbering scheme that also encodes the level of the pixel.
skymap	str		
tract	int		
patch	int		
label	str		A string value composed only of letters, numbers, and underscores.

TABLE 1: Dataset Columns

2.2 DatasetType

A DatasetType captures two properties of a Dataset and associates them with a string name:

- a *StorageClass*;
- a set of DataUnit keys whose corresponding values must be provided to uniquely identify a Dataset within a Collection.

StorageClasses are things that Datastores know how to serialize. They are closely related to the in-memory data structure or class object used by a Dataset, and in most cases they have a one-to-one relationship with those concepts. In other cases, a StorageClass may just correspond to an intermediate opaque serialization interface (e.g. Python's `pickle`) that can be used to store objects of many different types. Opaque StorageTypes generally severely limit the flexibility of Datastores to choose how objects are stored and make it impossible to retreive components or slices from those datasets, but they provide a way to save almost-arbitrary objects to Datastores without adding a new StorageClass each time. Because the full set of StorageClasses must in general be known to all Datastores, the set of supported StorageClasses and their definitions is maintained in code, not any particular Registry.

DatasetTypes are expected to be much more dynamic than StorageClasses or DataUnits; developers should be able to easily define new DatasetTypes by mixing an existing StorageClass with a set of existing DataUnits and a new name. We nevertheless expect DatasetType creation to be quite rare compared with Dataset creation, and certain Registries may limit DatasetType creation to superusers or require adherence to strict naming conventions (see Section A.2).

The columns of the DatasetType table itself are shown in Table 2. The list of associated DataUnits is managed by the DatasetTypeUnits table, and a list of associated metadata tables (see Section 6) is managed by the DatasetTypeMetadata table.

Name	Type	Attributes	Description
dataset_type_name	str	PRIMARY KEY	Globally unique name for this DatasetType.
storage_class	str	NOT NULL	Name of the StorageClass associated with this DatasetType. All registries must support the full set of standard StorageClasses, so the set of allowed StorageClasses and their properties is maintained in the registry Python code rather than the database.

TABLE 2: DatasetType Columns

Name	Type	Attributes	Description
dataset_type_name	str	NOT NULL	The name of the DatasetType.
unit_name	str	NOT NULL	The name of a DataUnit associated with this DatasetType.

TABLE 3: DatasetTypeUnits Columns

Name	Type	Attributes	Description
dataset_type_name	str	NOT NULL	The name of the DatasetType.
metadata_name	str	NOT NULL	The name of a Metadata table that has a record for every Dataset entry with this DatasetType.

TABLE 4: DatasetTypeMetadata Columns

2.3 Composite Datasets

Datasets may be composite: they may contain discrete named child Datasets that can be retrieved efficiently from a parent or combined to form a new parent.

The structure of composite dataset is fully defined by its StorageClass; all Datasets with a particular StorageClass will have the same set of component names and component StorageClasses (though some StorageClasses may permit a component to be NULL).

When a DatasetType is registered with a StorageClass that has components, DatasetTypes for those components are automatically created as well. The entries in DatasetTypeUnits for these child DatasetTypes will be the same as those for the parent, and the names for the children will have the form `{parent-dataset-name}.{component-name}`.

Name	Type	Attributes	Description
parent_dataset_id	int	PRIMARY KEY	Link to the Dataset entry for the parent/composite dataset.
component_dataset_id	int	PRIMARY KEY	Link (with component_dataset_id) to the Dataset entry for a child/component dataset.
component_name	str	NOT NULL	Name of this component within this composite.

TABLE 5: DatasetComposition Columns

.

Both parent/composite datasets and child/component Datasets always have entries in the Dataset table, and these are related by the entries of the DatasetComposition table.

Some Datasets are *virtual composites*, which means that they are not directly stored in any Datastore. These are instead assembled entirely from their components using an "Assembler" function stored in the Dataset table's `assembler` field. Note that virtual composites are still "more than the sum of their parts" from a Registry perspective. They have their own entries in the Dataset table, a number of entries in the DatasetComposition table, and potentially entries in one or more metadata tables. This means that they must be explicitly created (though unlike other datasets, this can be done without a Datastore) before they can be retreived.

3 DataUnits

A DataUnit is a predefined discrete unit of data that can be used to label a Dataset, such as a Visit or Tract. Together, the set of DataUnits are the keys that may be used in data ID dicts,

but DataUnits can also be associated with additional metadata fields and other DataUnits. These relationships and metadata fields are predefined and managed by the Registry, and are hence a major part of the common Registry schema described by this document.

All of the concrete DataUnits described in this section play the same role in how they relate to Datasets, but they can have very different representations in the SQL schema.

All DataUnits have one or more **Value Fields**, which provide links to the Dataset table. A DataUnit's own Value Fields are not necessarily sufficient to uniquely identify its instances, however; DataUnits can have **Dependencies**, which are other DataUnits whose Value Fields must *also* be provided (recursively) for uniqueness.

Not all DataUnits have tables. Those that do have a (typically) compound primary key that includes its Value Fields and those of its Dependencies (again, recursively). A DataUnit table can also have a foreign key constraint that is not a Dependency; for example, a Visit has a foreign key to PhysicalFilter, but the PhysicalFilter is not part of Visit's compound primary key.

3.1 Fundamental DataUnits

Fundamental DataUnits are those that are not associated with a Camera or SkyMap.

3.1.1 Label

An arbitrary string value. There is no SQL representation or constraint on the values a Label can take. Multiple labels are represented (by convention) by a comma-separated string.

Dependencies: none

Value Fields:

- **label (str):** A string value composed only of letters, numbers, and underscores.

Table: none

3.1.2 AbstractFilter

A filter that is not associated with a particular Camera. An abstract filter can be used to relate similar physical filters, and is typically the filter associated with coadds.

Dependencies: none

Value Fields:

- **abstract_filter (str):** String name for the abstract filter, frequently a single character.

Table: AbstractFilter

Name	Type	Attributes	Description
abstract_filter	str	PRIMARY KEY	Name of the filter.

TABLE 6: AbstractFilter Columns

3.1.3 SkyPix

A pixel in a hierarchical decomposition of the sky (e.g. HTM, Q3C, or HEALPix; we will select and support just one, but which is TBD). Has no SQL representation; even a definition table is not necessary, given that the allowable values and the associated spatial regions are best computed on-the-fly. SkyPix units are preferred to SkyMap (i.e. Tract- Patch) units for Datasets without any overlap regions (e.g. sharded reference catalogs). There are also considerable advantages to standardizing on just one level of the standard pixelization: if all SkyPix values are at a single level, they can be indexed using standard B-Trees and compared with simple equality comparison. In contrast, comparing SkyPix values at different levels requires pixelization- specific bit-shifting operations and custom indexes, which are much harder to implement across multiple RDMSs. As a result, we will (at least initially) try to define just a single level for all SkyPix values. Our preliminary guess is that this level should have pixels be approximately (within a factor of ~4) of the size of a single Sensor on the sky.

Dependencies: none

Value Fields:

- **skypix (int):** Unique id of a pixel in the hierarchical pixelization, using a numbering scheme that also encodes the level of the pixel.

Table: none

3.2 Camera DataUnits

Camera DataUnits are associated with a particular observatory and instrument, and are generally customized by a particular obs package.

Some Camera DataUnits are populated when a Camera is first defined within a Registry (Camera, PhysicalFilter, Sensor), while others are created when observations are ingested (Exposure, Visit).

ExposureRanges are unique among Camera DataUnits in being defined directly by the existence of one or more Datasets that use them. All other Camera DataUnits have their own tables that contain entries that are independent of any particular Dataset (and are typically each associated with many Datasets).

Each combination of Visit and Sensor is also associated with an entry in another table, VisitSensorRegion, which holds the spatial region on the sky associated with each such combination.

3.2.1 Camera

An entity that produces observations. A Camera defines a set of PhysicalFilters and Sensors and a numbering system for the Exposures and Visits that represent observations with it.

Dependencies: none

Value Fields:

- **camera (str):** Globally unique string indentifier for this Camera.

Table: Camera

Name	Type	Attributes	Description
camera	str	PRIMARY KEY	Globally unique string indentifier for this Camera.

TABLE 7: Camera Columns

3.2.2 PhysicalFilter

A filter associated with a particular Camera. PhysicalFilters are used to identify datasets that can only be associated with a single observation.

Dependencies: Camera

Value Fields:

- **physical_filter (str):** String name of the filter, typically a multi-letter code in a convention defined by the Camera (e.g. ”HSC-I” or ”F775W”).

Table: PhysicalFilter

Name	Type	Attributes	Description
camera	str	PRIMARY KEY	Name of the Camera with which this filter is associated.
physical_filter	str	PRIMARY KEY	String name of the filter, typically a multi-letter code in a convention defined by the Camera (e.g. ”HSC-I” or “F775W”).
abstract_filter	str		Name of the AbstractFilter with which this filter is associated.

TABLE 8: PhysicalFilter Columns

3.2.3 Sensor

A sensor associated with a particular Camera (not an observation of that sensor; that requires specifying an exposure or visit as well).

Dependencies: Camera

Value Fields:

- **sensor (str):** A unique (with Camera) integer identifier for the Sensor.

Table: Sensor

3.2.4 Exposure

An observation associated with a particular camera. All direct observations are identified with an Exposure, but derived datasets that may be based on more than one Exposure (e.g. multiple snaps) are typically identified with Visits instead, even for cameras that don’t have multiple Exposures per Visit. As a result, Cameras that don’t have multiple Exposures per Visit will typically have Visit entries that are essentially duplicates of their Exposure entries.

Name	Type	Attributes	Description
camera	str	PRIMARY KEY	Name of the Camera with which this Sensor is associated.
sensor	int	PRIMARY KEY	A unique (with Camera) integer identifier for the Sensor.
name	str		An alternate string identifer for the sensor; may or may not be unique within a Camera.
group	str		A string name for a group of sensors with a Camera-dependent interpretation, such as LSST's rafts.
purpose	str		Role of the Sensor; typically one of "SCIENCE", "WAVEFRONT", or "GUIDE", though Cameras may define additional values.

TABLE 9: Sensor Columns

The Exposure table contains metadata entries that are relevant for calibration Exposures, and does not duplicate entries in Visit that would be the same for all Exposures within a Visit.

Dependencies: Camera

Value Fields:

- **exposure (int):** Unique (with camera) integer identifier for this Exposure.

Table: Exposure

Name	Type	Attributes	Description
camera	str	PRIMARY KEY	The Camera used to observe the Exposure.
exposure	int	PRIMARY KEY	Unique (with camera) integer identifier for this Exposure.
visit	int		ID of the Visit this Exposure is associated with. Science observations should essentially always be associated with a visit, but calibration observations may not be.
physical_filter	str	NOT NULL	The bandpass filter used for all exposures in this Visit.
snap	int		If visit is not null, the index of this Exposure in the Visit, starting from zero.
datetime_begin	datetime		TAI timestamp of the start of the Exposure.
datetime_end	datetime		TAI timestamp of the end of the Exposure.
exposure_time	float		Duration of the Exposure with shutter open (seconds).
dark_time	float		Duration of the Exposure with shutter closed (seconds).

TABLE 10: Exposure Columns

3.2.5 Visit

A sequence of observations processed together, comprised of one or more Exposures from the same Camera with the same pointing and PhysicalFilter. The Visit table contains metadata that is both meaningful only for science Exposures and the same for all Exposures in a Visit.

Dependencies: Camera

Value Fields:

- **visit (int):** Unique (with camera) integer identifier for this Visit.

Table: Visit

Name	Type	Attributes	Description
camera	str	PRIMARY KEY	The Camera used to observe the Exposures associated with this Visit.
visit	int	PRIMARY KEY	Unique (with camera) integer identifier for this Visit.
physical_filter	str	NOT NULL	The bandpass filter used for all exposures in this Visit.
datetime_begin	datetime		TAI timestamp of the beginning of the Visit. This should be the same as the datetime_begin of the first Exposure associated with this Visit.
datetime_end	datetime		TAI timestamp of the end of the Visit. This should be the same as the datetime_end of the last Exposure associated with this Visit.
exposure_time	float		The total exposure time of the Visit in seconds. This should be equal to the sum of the exposure_time values for all constituent Exposures (i.e. it should not include time between Exposures).
seeing	float		Average seeing, measured as the FWHM of the Gaussian with the same effective area (arcsec).
region	bytes		A spatial region on the sky that bounds the area covered by the Visit. This is expected to be more precise than the region covered by the SkyPixels associated with the Visit, but may still be larger than the Visit as long as it fully covers it. Must also fully cover all regions in the VisitSensorRegion entries associated with this Visit. Regions are lsst.sphgeom.ConvexPolygon objects persisted as portable (but not human-readable) bytestrings using the encode and decode methods.

TABLE 11: Visit Columns

3.2.6 ExposureRange

An inclusive range of Exposures that may be open in either direction, typically used to identify master calibration products. There is no SQL table associated with ExposureRanges; there is no additional information associated with an ExposureRange besides the camera, valid_first, and valid_last fields already present in Dataset.

Dependencies: Camera

Value Fields:

- **valid_first (int):** First exposure identifier included in the range (inclusive). may be zero to indicate an open interval.
- **valid_last (int):** Last exposure identifier included in the range (inclusive). may be max(int) to indicate an open interval.

Table: none

3.3 SkyMap DataUnits

SkyMap DataUnits together define a two-level subdivision of the sky with overlaps, suitable for coaddition and coadd processing.

3.3.1 SkyMap

A set of Tracts and Patches that subdivide the sky into rectangular regions with simple projections and intentional overlaps.

Dependencies: none

Value Fields:

- **skymap (str):** A human-readable name for the SkyMap, used as its unique identifier.

Table: SkyMap

Name	Type	Attributes	Description
skymap	str	PRIMARY KEY	A human-readable name for the SkyMap, used as its unique identifier.
hash	str	NOT NULL	A hash of the SkyMap's parameters, used to prevent duplicate SkyMaps with the different names from being registered.

TABLE 12: SkyMap Columns

3.3.2 Tract

A large rectangular region mapped to the sky with a single map projection, associated with a particular SkyMap.

Dependencies: SkyMap

Value Fields:

- **tract (int):** Unique (with SkyMap) integer identifier for the Tract.

Table: Tract

Name	Type	Attributes	Description
skymap	str	PRIMARY KEY	The SkyMap with which this Tract is associated.
tract	int	PRIMARY KEY	Unique (with SkyMap) integer identifier for the Tract.
ra	float		Right ascension of the center of the tract (degrees).
dec	float		Declination of the center of the tract (degrees).
region	bytes		A spatial region on the sky that bounds the area associated with the Tract. This is expected to be more precise than the SkyPixels associated with the Visit (see TractSkyPixJoin), but may still be larger than the Tract as long as it fully covers it. Regions are lsst.sphgeom.ConvexPolygon objects persisted as portable (but not human-readable) bytestrings using the encode and decode methods.

TABLE 13: Tract Columns

3.3.3 Patch

A rectangular region within a Tract.

- Tract
- SkyMap

Value Fields:

- **patch (int):** Unique (with SkyMap and Tract) integer identifier for the Patch.

Table: Patch

3.4 Joins Between DataUnits

Many predefined DataUnit relationships are many-to-many, and hence are not captured in the descriptions of individual DataUnits above. Some of these relationships can be implemented

Name	Type	Attributes	Description
skymap	str	PRIMARY KEY	The SkyMap with which this Patch is associated.
tract	int	PRIMARY KEY	The Tract with which this Patch is associated.
patch	int	PRIMARY KEY	Unique (with SkyMap and Tract) integer identifier for the Patch.
cell_x	int	NOT NULL	Which column this Patch occupies in the Tract's grid of Patches.
cell_y	int	NOT NULL	Which row this Patch occupies in the Tract's grid of Patches.
region	bytes		A spatial region on the sky that bounds the area associated with the Patch. This is expected to be more precise than the SkyPixels associated with the Visit (see PatchSkyPixJoin), but may still be larger than the Patch as long as it fully covers it. Regions are lsst.sphgeom.ConvexPolygon objects persisted as portable (but not human-readable) bytestrings using the encode and decode methods.

TABLE 14: Patch Columns

as join tables or views, but others are just SQL expressions that can be used in a SELECT statement's JOIN clause.

The complete set of conceptual DataUnit relationships is shown in Figure 3.

3.4.1 VisitSensorRegion

A many-to-many join table that provides region information for Visit-Sensor combinations.

Table: VisitSensorRegion

Name	Type	Attributes	Description
camera	str	PRIMARY KEY	Name of the Camera associated with the Visit and Sensor.
visit	int	PRIMARY KEY	Visit ID
sensor	int	PRIMARY KEY	Sensor ID
region	bytes		A spatial region on the sky that bounds the area associated with this Visit+Sensor combination. This is expected to be more precise than the SkyPixels associated with the Visit+Sensor (see VisitSensorSkyPixJoin), but may still be larger than the true region as long as it fully covers it. Regions are lsst.sphgeom.ConvexPolygon objects persisted as portable (but not human-readable) bytestrings using the encode and decode methods.

TABLE 15: VisitSensorRegion Columns

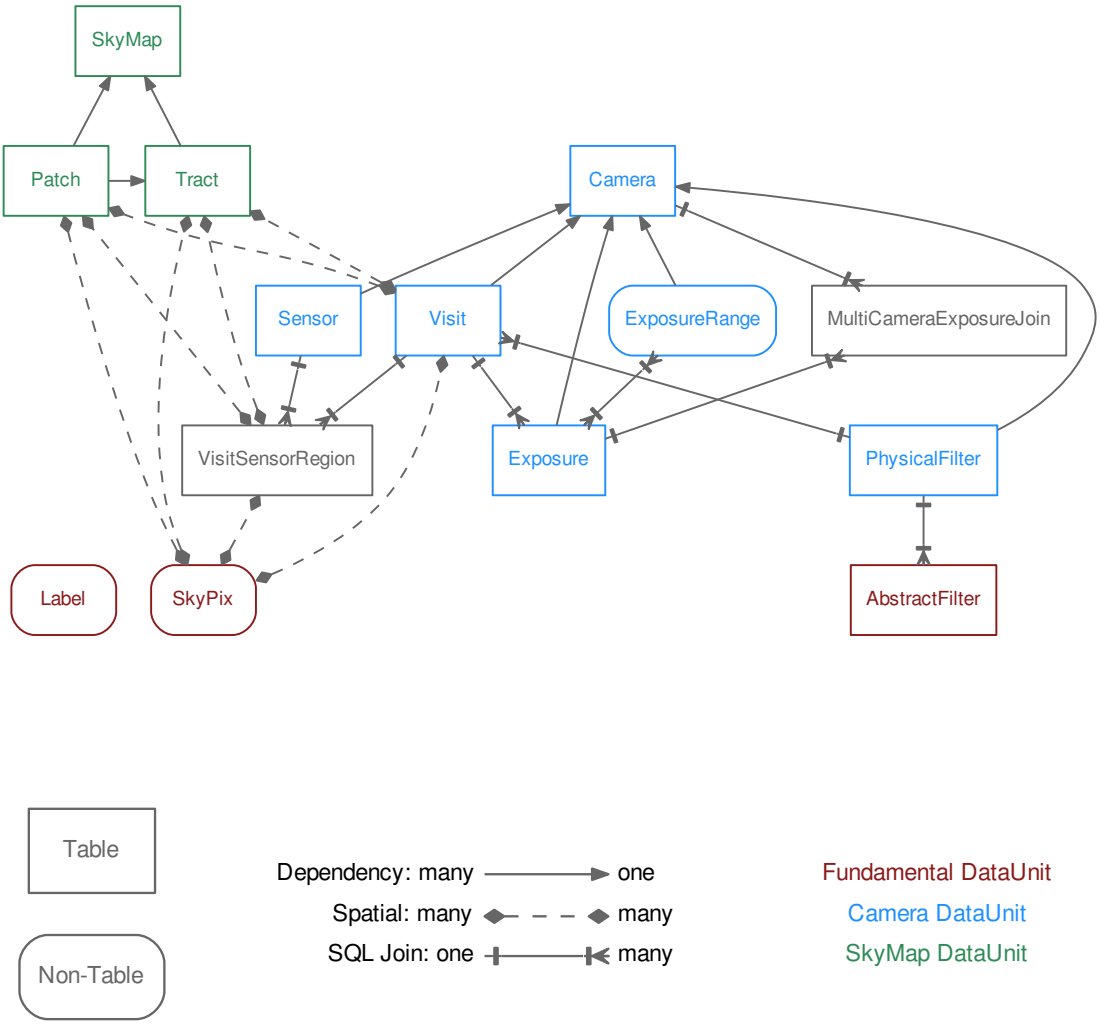

FIGURE 3: DataUnit Conceptual Joins

3.4.2 ExposureRangeJoin

A calculated join between Datasets identified with an Exposure (typically raw science frames) and Datasets identified with ExposureRange (typically master calibrations).

3.4.3 MultiCameraExposureJoin

A join table that relates Exposures from different Cameras, with explicitly-created entries. This is intended to be used primarily in calibration products production, to relate auxilliary telescope observations with the main camera observations they support. It could also be used to relate CBP state (represented as CBP "Exposures") with actual main camera observations of the CBP.

Table: MultiCameraExposureJoin

Name	Type	Attributes	Description
camera_1	str	NOT NULL	Camera name for lhs Dataset.
exposure_1	int	NOT NULL	Exposure ID for lhs Dataset.
camera_2	str	NOT NULL	Camera name for rhs Dataset.
exposure_2	int	NOT NULL	Exposure ID for rhs Dataset.

TABLE 16: MultiCameraExposureJoin Columns

3.4.4 VisitSensorSkyPixJoin

A spatial join table that relates Visit+Sensor to SkyPix, also used as an intermediate when relating Visit+Sensor to SkyMap DataUnits. Entries are expected to be calculated outside the database and added/updated whenever VisitSensorRegion is.

Table: VisitSensorSkyPixJoin

Name	Type	Attributes	Description
camera	str	NOT NULL	Name of the Camera associated with the Visit and Sensor.
visit	int	NOT NULL	Visit ID
sensor	int	NOT NULL	Sensor ID
skypix	int	NOT NULL	ID of a SkyPix that overlaps the Visit+Sensor combination.

TABLE 17: VisitSensorSkyPixJoin Columns

3.4.5 VisitSkyPixJoin

A spatial join table that relates Visitto SkyPix, also used as an intermediate when relating Visit to SkyMap DataUnits. This can and probably should be implemented as a trivial view on VisitSensorSkyPixJoin.

View: VisitSkyPixJoin, defined as:

```
SELECT DISTINCT camera, visit, skypix FROM VisitSensorSkyPixJoin;
```

Name	Type	Attributes	Description
camera	str	NOT NULL	Name of the Camera associated with the Visit.
visit	int	NOT NULL	Visit ID
skypix	int	NOT NULL	ID of a SkyPix that overlaps the Visit combination.

TABLE 18: VisitSkyPixJoin Columns

3.4.6 PatchSkyPixJoin

A spatial join table that relates Patch to SkyPix, also used as an intermediate when relating Patch to Camera DataUnits. Entries are expected to be calculated outside the database and added along with the Patch itself.

Table: PatchSkyPixJoin

Name	Type	Attributes	Description
skymap	str	NOT NULL	Name of the SkyMap associated with the Patch.
tract	int	NOT NULL	Tract ID
patch	int	NOT NULL	Patch ID
skypix	int	NOT NULL	ID of a SkyPix that overlaps the Patch.

TABLE 19: PatchSkyPixJoin Columns

3.4.7 TractSkyPixJoin

A spatial join table that relates Tract to SkyPix, also used as an intermediate when relating Tract to Camera DataUnits. This can and probably should be implemented as a trivial view on PatchSkyPixJoin.

View: TractSkyPixJoin, defined as:

```
SELECT DISTINCT skymap, tract, skypix FROM PatchSkyPixJoin;
```

Name	Type	Attributes	Description
skymap	str	NOT NULL	Name of the SkyMap associated with the Tract.
tract	int	NOT NULL	Tract ID
skypix	int	NOT NULL	ID of a SkyPix that overlaps the Tract.

TABLE 20: TractSkyPixJoin Columns

3.4.8 VisitSensorPatchJoin

A spatial join table that relates Visit+Sensor to Patch via SkyPix. Should be implemented as a view; it may be materialized as long as it can be kept up to date when new Visits or SkyMaps are added. If a database UDF is available to determine whether two regions overlap, we could include that in this view to refine the results. For now, we will assume that such a UDF is not available.

View: VisitSensorPatchJoin, defined as:

```
  SELECT DISTINCT
  VisitSensorSkyPixJoin.camera,
  VisitSensorSkyPixJoin.visit,
  VisitSensorSkyPixJoin.sensor,
  PatchSkyPixJoin.skymap,
  PatchSkyPixJoin.tract,
  PatchSkyPixJoin.patch
FROM
  VisitSensorSkyPixJoin INNER JOIN PatchSkyPixJoin ON (
    VisitSensorSkyPixJoin.skypix = PatchSkyPixJoin.skypix
  );
```

3.4.9 VisitPatchJoin

A spatial join table that relates Visit to Patch via SkyPix. Should be implemented as a view; it may be materialized as long as it can be kept up to date when new Visits or SkyMaps are added. If a database UDF is available to determine whether two regions overlap, we could

Name	Type	Attributes	Description
camera	str	NOT NULL	Name of the Camera associated with the Visit and Sensor.
visit	int	NOT NULL	Visit ID
sensor	int	NOT NULL	Sensor ID
skymap	str	NOT NULL	Name of the SkyMap associated with the Patch.
tract	int	NOT NULL	Tract ID
patch	int	NOT NULL	Patch ID

TABLE 21: VisitSensorPatchJoin Columns

include that in this view to refine the results. For now, we will assume that such a UDF is not available.

View: VisitPatchJoin, defined as:

```
  SELECT DISTINCT
  VisitSkyPixJoin.camera,
  VisitSkyPixJoin.visit,
  PatchSkyPixJoin.skymap,
  PatchSkyPixJoin.tract,
  PatchSkyPixJoin.patch
FROM
  VisitSkyPixJoin INNER JOIN PatchSkyPixJoin ON (
    VisitSkyPixJoin.skypix = PatchSkyPixJoin.skypix
  );
```

Name	Type	Attributes	Description
camera	str	NOT NULL	Name of the Camera associated with the Visit.
visit	int	NOT NULL	Visit ID
skymap	str	NOT NULL	Name of the SkyMap associated with the Patch.
tract	int	NOT NULL	Tract ID
patch	int	NOT NULL	Patch ID

TABLE 22: VisitPatchJoin Columns

3.4.10 VisitSensorTractJoin

A spatial join table that relates Visit+Sensor to Tract via SkyPix. Should be implemented as a view; it may be materialized as long as it can be kept up to date when new Visits or SkyMaps are added. If a database UDF is available to determine whether two regions overlap, we could

include that in this view to refine the results. For now, we will assume that such a UDF is not available.

View: VisitSensorTractJoin, defined as:

```
  SELECT DISTINCT
  VisitSensorSkyPixJoin.camera,
  VisitSensorSkyPixJoin.visit,
  VisitSensorSkyPixJoin.sensor,
  TractSkyPixJoin.skymap,
  TractSkyPixJoin.tract
FROM
  VisitSensorSkyPixJoin INNER JOIN TractSkyPixJoin ON (
    VisitSensorSkyPixJoin.skypix = TractSkyPixJoin.skypix
  );
```

Name	Type	Attributes	Description
camera	str	NOT NULL	Name of the Camera associated with the Visit and Sensor.
visit	int	NOT NULL	Visit ID
sensor	int	NOT NULL	Sensor ID
skymap	str	NOT NULL	Name of the SkyMap associated with the Tract.
tract	int	NOT NULL	Tract ID

TABLE 23: VisitSensorTractJoin Columns

3.4.11 VisitTractJoin

A spatial join table that relates Visit to Tract via SkyPix. Should be implemented as a view; it may be materialized as long as it can be kept up to date when new Visits or SkyMaps are added. If a database UDF is available to determine whether two regions overlap, we could include that in this view to refine the results. For now, we will assume that such a UDF is not available.

View: VisitTractJoin, defined as:

```
  SELECT DISTINCT
  VisitSkyPixJoin.camera,
  VisitSkyPixJoin.visit,
```

```
  TractSkyPixJoin.skymap,
  TractSkyPixJoin.tract
FROM
  VisitSkyPixJoin INNER JOIN TractSkyPixJoin ON (
    VisitSkyPixJoin.skypix = TractSkyPixJoin.skypix
  );
```

Name	Type	Attributes	Description
camera	str	NOT NULL	Name of the Camera associated with the Visit.
visit	int	NOT NULL	Visit ID
skymap	str	NOT NULL	Name of the SkyMap associated with the Tract.
tract	int	NOT NULL	Tract ID

TABLE 24: VisitTractJoin Columns

4 Collections and Provenance

4.1 Collections

A Collection is a group of Datasets that is constrained to have at most one Dataset for any combination of DatasetType and identifying DataUnits. The inputs and outputs of a particular processing run is typically associated with a Collection, as are human-curated combinations of related processing runs.

Only one Collection within a Registry is accessible to any Butler client, making them from a user perspective the most natural analog to the Gen2 Butler's Data Repositories. The constraint on Dataset uniqueness within a Collection ensures that any `Butler.get` call has an unambiguous result. Instead of lazily chaining Collections in the manner of Gen2 Data Repositories, we simply permit Datasets to belong to multiple Collections.

Collections are implemented via a simple tag table, DatasetCollection, whose entries are just tuples of a `dataset_id` and the string name that identifies the Collection.

It may be necessary for performance reasons to introduce an integer primary key for Collections, along with a table to relate each Collection name to its primary key (and possibly other metadata). Doing this now seems premature.

Name	Type	Attributes	Description
dataset_id	int	PRIMARY KEY	Link to a unique record in the Dataset table.
collection	str	PRIMARY KEY	Name of a Collection with which this Dataset is associated.

TABLE 25: DatasetCollection Columns

4.2 Execution

Records of the Execution table can be used to represent any step in a production. Executions are frequently associated one-to-one with records of other tables that use the same primary key fields (such as Run and Quantum). Conceptually these tables are Execution specializations. Executions themselves only record basic information about the step and cannot be directly nested.

The full set of Execution columns can be found in Table 26.

Name	Type	Attributes	Description
execution_id	int	PRIMARY KEY	A unique autoincrement field used as the primary key for Execution.
start_time	datetime		The start time for the execution. May have a different interpretation for different kinds of execution.
end_time	datetime		The end time for the execution. May have a different interpretation for different kinds of execution.
host	str		The system on which the execution was run. May have a different interpretation for different kinds of execution.

TABLE 26: Execution Columns

4.3 Run

Run is a specialization of Execution used to capture coarse provenance. Every Dataset and Quantum must be associated with a Run.

For Datasets produced by SuperTask Pipelines, a Run represents an execution of a single Pipeline with no change to its configuration or the software environment. Other special Runs may represent raw data ingest mechanisms.

The full set of Run columns can be found in Table 27.

4.4 Quantum

Quantum is a specialization of Execution used to capture fine-grained provenance for Datasets produced by SuperTasks.

Name	Type	Attributes	Description
execution_id	int	PRIMARY KEY	A unique integer identifier for this Run, which is also the execution_id for an associated Execution record.
collection	str		A Collection name with which all Datasets in this Run are initially associated, also used as a human-readable name for this Run.
environment_id	int		A dataset_id linking to a Dataset that contains a description of the software environment (e.g. versions) used for this Run.
pipeline_id	int		A dataset_id linking to a Dataset that contains a serialization of the SuperTask Pipeline used for this Run (if any).

TABLE 27: Run Columns

Each Quantum record is uniquely associated with an Execution record.

The full set of Quantum columns can be found in Table 28.

Name	Type	Attributes	Description
execution_id	int	PRIMARY KEY	A unique integer identifier for this Quantum, which is also the execution_id for an associated Execution record.
task	str		Fully-qualified name of the SuperTask that executed this Quantum.
run_id	int		Link to the Run this Quantum is a part of.

TABLE 28: Quantum Columns

5 Datastore Information

The DatasetStorage table provides public information about how Datasets are stored in particular Datastores. This includes whether they are present at all, which is indicated by the existence of a record with a particular Dataset/Datastore combination.

Name	Type	Attributes	Description
dataset_id	int	PRIMARY KEY	Link to the Dataset table.
datastore_name	str	PRIMARY KEY	Name of the Datastore this entry corresponds to.

TABLE 29: DatasetStorage Columns

This table is unique among Registry tables in that it is updated directly by Datastore, rather than via Butler (whether this goes through a Registry client or some other common interface to the database is TBD). In general, Datastores must also record private information about each Dataset (e.g. filenames, read formatters). These entries may be stored in additional tables in the same database that holds the Registry, but may also be stored elsewhere, and are never considered part of the Registry even if they are stored in the same database.

6 Additional Metadata Tables

Registries may have additional metadata tables that are associated only with certain Dataset or DataUnit entries. These fall into three categories:

- metadata for Datasets with a particular StorageClass;
- metadata for Datasets with a particular DatasetType;
- metadata for Camera DataUnits associated with a particular Camera.

No metadata tables are required for a fully-functional Butler implementation, but we expect them to contain quantities that may be frequently used in user queries to locate Datasets with certain properties (including the expressions used to describe what processing should be performed by the SuperTask framework).

While the set of StorageClasses is predefined and shared by all Registries, it is currently considered out of scope for this document, and hence the metadata tables are as well. We nevertheless expect the set of StorageClass-specialized Dataset metadata tables to be common to all Registries, and a future version of this document will include a complete description of their schemas. StorageClass-specific metadata tables must have `dataset_id` field that is used both as that table's primary key and a foreign key to the Dataset table.

The set of DatasetTypes is very much dynamic, and may not be the same for different Registries. DatasetType-specific Dataset metadata tables are thus never expected to be a part of the common schema, and expressions that rely on them are always considered non-portable. DatasetType-specific metadata tables must have `dataset_id` field that is used both as that table's primary key and a foreign key to the Dataset table.

While not all Registries will have the full set of Cameras, any Registry that contains a certain Camera can be expected to have all of the custom metadata tables associated with it. Expressions that rely on Camera-specific metadata tables are obviously not portable between Cameras, and for this reason Camera specialization code should attempt to put most frequently-used metadata in the generic fields provided by the generic Camera DataUnit tables (e.g. Sensor.group) and avoid any need for Camera-specific metadata tables. The definition of the schemas of Camera-specific DataUnit metadata tables is delegated to the `obs` package responsible for defining that camera.

6.1 POSIX Filesystem Datastores

A Datastore backed by a POSIX filesystem is expected to be a major component of the LSST production environment. It also serves as a useful example of the kind of tabular information Datastores must store outside the Registry.

A POSIX Filesystem Datastore is configured[1] by associating a *Formatter* object with each StorageClass. Formatters having methods for both reading and writing objects, but the configuration is only used to select the Formatter used when writing. When a Dataset is written, the name of the Formatter is saved by the Datastore and associated with that Dataset, so the same Formatter can be used for reading later. This means that instances of a single StorageClass can be stored in multiple ways within a single Datastore, and the user need not know the configuration that was used to write a Dataset in order to read it.

The filesystem paths in a POSIX Filesystem Datastore are computed relative to the root of the data repository, and must be unique for each Dataset. This may be achieved by inserting the Dataset's Run ID and DataUnit values into a string template that is unique for its DatasetType; this produces paths of the same form as those of the Gen2 Butler. Unique filenames can also be achieved smply by including the primary key of the 1 table (`dataset_id`). Our implementation permits but does not require filesystem paths to include subdirectories.

The internal information a POSIX Filesystem Datastore records for each Dataset is thus:

- `dataset_id` (integer, primary key);
- the Formatter name (string);
- the filesystem path (string);
- the StorageClass name (string).

Our implementation currently uses special StorageClass names to indicate certain types of composite dataset storage, which means that the Datastore cannot rely on obtaining the StorageClass from the Butler. This may change in the future, but it may continue to be useful to record the StorageClass here as well as a consistency check.

As noted above, this internal information may be stored either within the same database as a Registry or in an external (not necessarily SQL) database.

[1] in part; a full discussion of Datastore configuration is beyond the scope of this document

A Possible Modifications for Multi-User Environments

Some database servers will be expected to effectively handle multiple layered Registries, or at least provide a single Registry with complex, multi-user Dataset ownership. For example, a Registry that supports test processing for internal LSST development could allow all users to see all entries in all tables, but users could be permitted to create and modify only entities (e.g. Collections, Datasets, DataUnits) that they created. The Registries that support the public LSST Science Platform must be considerably more complex; each user effectively sees a different Registry, because in addition to the common (and read-only) official LSST data products they may have read and possibly write access to different user-produced data products. These cannot be different databases; user-driven processing will essentially always use official data products as inputs, and user-defined Collections will certainly include official Datasets.

These features are probably best implemented in different ways for different RDBMSs, but they should build on similar functionality for layered sharing of astronomical catalogs that has long been planned for the Science Platform (and already exists in other astronomical database systems, such as SkyServer's CasJobs). The number of entries in multi-user Registry tables should be orders of magnitude smaller than the the number of entries in shared astronomical catalogs (even the smallest catalog Datasets typically have thousands of records but require just a handful of Registry entries), but Registy tables may have much more complex relationships.

One technique for supporting layered Registries that may be useful for any RDBMS is splitting up a Registry "table" into multiple actualy tables (for different users, groups, or access levels) and providing a (possibly temporary) view that combines them. For example, assuming an RDBMS that supports namespaces of some kind, we could provide per-user Collections with something like the following:

```
% When creating the data release:
CREATE TABLE dr2.DatasetCollections (dataset_id int, collection string);

% ...populate dr2 tables...

% When user Alice signs up for a personal data repository:
CREATE TABLE users.alice.DatasetCollections (dataset_id, collection string);

% Whenever Alice creates a Registry client:
```

```
CREATE TEMPORARY VIEW DatasetCollection
    SELECT * FROM dr2.DatasetCollection
    UNION
    SELECT * FROM users.alice.DatasetCollection;
```

We have explicitly declared the common schema to be a read-only (i.e. SELECT-only) interface for precisely this reason; all operations that need to update or append to common schema "tables" must go through a Python interface that can be customized by specific Registry implementations to write to the appropriate underlying tables.

A.1 Cross-Registry Auto-Increment Keys

This technique of using union views to combine multiple implementation tables is problematic for tables that have an autoincrement primary key, such as Dataset and Execution: inserts to different implementation tables could easily generate ID conflicts in the union.

Some RDBSs may provide ways for multiple tables to share an autoincrement ID source, which would of course be the most natural solution. ID ranges may also be reserved for different users, and could even be shared by users and rewritten only in the (probably rare) case where users with the same ID range opt to publish data to each other.

Another approach is to augment each autoincrement field with another field that is unique to each implementation table, and use them together as a compound primary key. This of course changes the public registy schema, and in fact an earlier version of the common schema included these fields. We have since dropped them after determining that they would not generally be useful in implementing transfers between Registries (e.g. between SQLite Registries on shared-nothing worker nodes and the Data Backbone), as other factors would make rewriting the IDs almost inevitable in those contexts. The compound primary key approach may be more useful in multi-user single-database Registries however, where ID rewriting is more disruptive and we have more control over the assignment of the per-table unique values. Adding compound primary keys back into the schema could be very disruptive if done after single-key Registries are in common use, however, so it is important to decide during initial Butler development if other approaches can be used instead.

A.2 Namespaces for String Keys

Some Tables and DataUnits use strings as unique identifiers, including DatasetType, Collections, SkyMaps, and Labels. These can also have problems with clashes in multi-user environments, regardless of whether the Registry uses union views to combine multiple implementation tables. For example, because the definition of a DatasetType cannot change as long as it has any Datasets associated with it, it may be prudent to use DatasetType names in operations that encode the name of the data release it is intended for. During construction, a cycle name or other periodically-refreshed namespace could be used for shared DatasetTypes (etc.) instead.

While normal users of a multi-user will most frequently use predefined, shared DatasetTypes, SkyMaps, and Labels, user-defined versions of these must be possible, and per-user Collections are expected to be quite common. These should probably include the username (probably as a prefix).

These conventions can of course be implemented within regular single-strings fields with no special database support, but it is worth considering whether it would be better to split the namespaces into separate fields. If the Python API provides a "using declaration" functionality that sets the default namespaces to be searched for quantities, having a separate namespace field for string keys might cut down on the need to do string manipulation in queries. It could also make it easier to implement automatic/enforced per-user namespaces when using union joins over multiple implementation tables. Again using Collections as an example, we could omit the Collection namespaces from the implementation tables, and only include them in the join:

```
% When creating the data release:
CREATE TABLE dr2.DatasetCollections (dataset_id int, collection string);

% ...populate dr2 tables...

% When user Alice signs up for a personal data repository:
CREATE TABLE users.alice.DatasetCollections (dataset_id, collection string);

% Whenever Alice creates a Registry client:
CREATE TEMPORARY VIEW DatasetCollection
    SELECT
```

```
    dr2.DatasetCollection.dataset_id AS dataset_id,
    "dr2" AS namespace,
    dr2.DatasetCollection.collection AS collection,
FROM dr2.DatasetCollection

UNION

SELECT
    users.alice.DatasetCollection.dataset_id AS dataset_id,
    "alice" AS namespace,
    users.alice.DatasetCollection.collection AS collection,
FROM users.alice.DatasetCollection;
```

Just like compound primary keys for tables with auto-increment keys, adding namespace fields to unique string identifiers could be highly disruptive if done after Registries are already in broad use. If this is considered a useful change, we should make it as early as possible in Butler development.