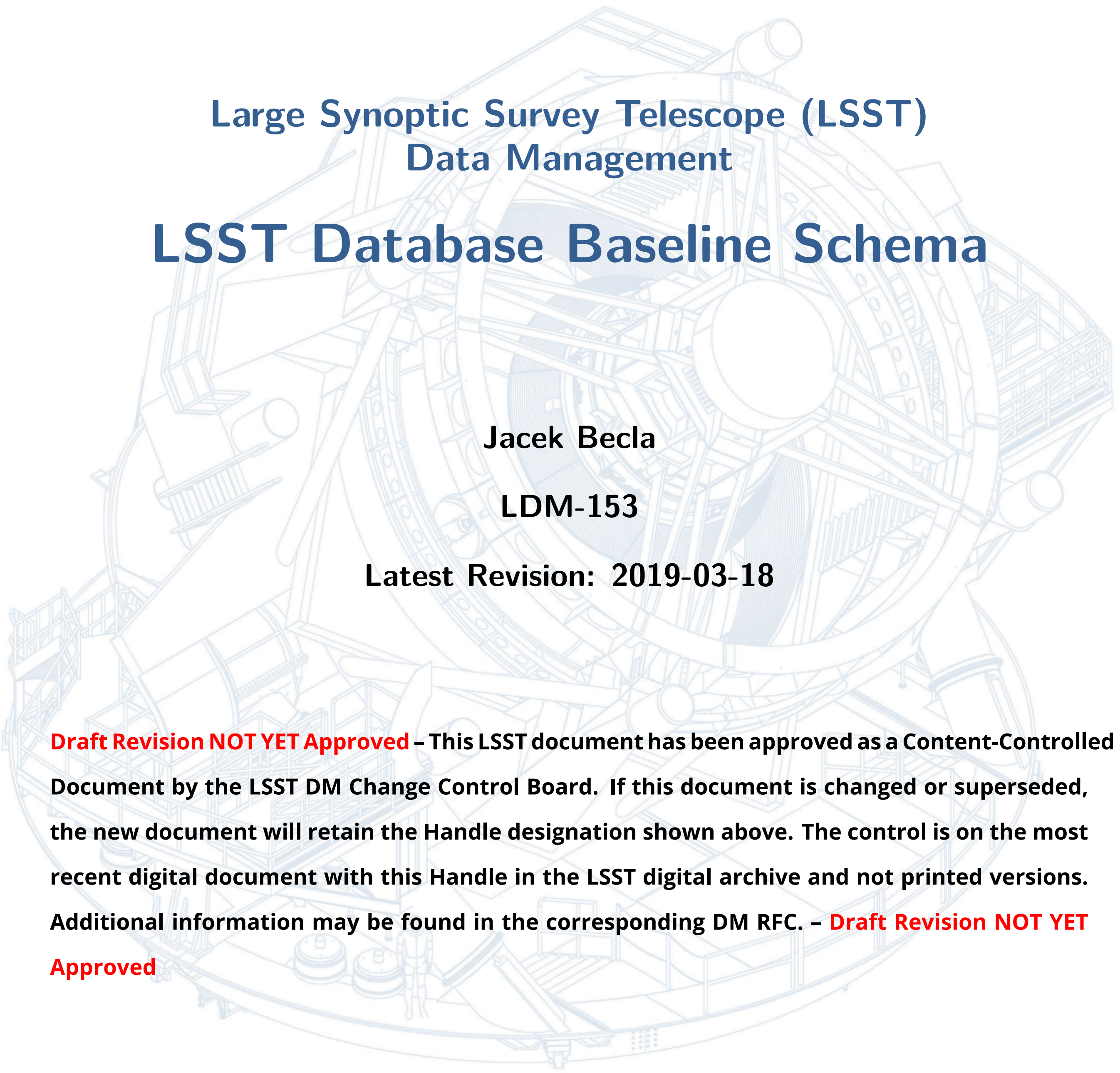
LARGE SYNOPTIC SURVEY TELESCOPE



Large Synoptic Survey Telescope (LSST)
Data Management

LSST Database Baseline Schema

Jacek Becla

LDM-153

Latest Revision: 2019-03-18

Change Record

Version	Date	Description	Owner name
1	2007-01-12	Initial Version.	Jacek Becla
2	2007-02-14	General edits.	Jacek Becla
3	2007-02-21	General edits.	Jacek Becla
4	2007-02-27	General edits.	Jacek Becla
5	2007-03-05	General edits.	Jacek Becla
6	2007-03-07	General edits.	Jacek Becla
7	2007-04-09	General edits.	Jacek Becla
8	2007-09-05	General edits.	Jacek Becla
9	2011-07-04	General edits.	Jacek Becla
10	2011-07-14	General edits.	Jacek Becla
11	2013-08-02	Major refresh of the schema to reflect science requirements described in Data Products doc. Added sections about schema evolution, provenance	Jacek Becla, Kian-Tat Lim
12	2013-09-30	Resynchronized with the latest Data Products Definition Document (v Sep 23 2013) and corresponding schema.	Jacek Becla
13	2013-10-10	TCT approved	R. Allsman
14	2013-10-18	Fixed Table of Contents	Jacek Becla

Contents

LSST Database Baseline Schema

1 Schema Format and Documentation

The LSST Database schema master files are stored in ASCII files and version-controlled (git) together with the rest of the LSST software. Each major schema version has its own dedicated file. We currently have a baseline schema, and 6 different versions of schemas for different data challenges: DC3a, PT1.1, PT1.2, ImSim, S12_sdss, and S12_lsstsim.

Each schema file contains a schema definition interleaved with rich table-level and column-level comments. Additional notes such as information about units are embedded. Comments follow a special HTML-like format that enables auto-generation of web-browsable pages. This approach maximizes vendor-independence of both the schema definition and its documentation. It enables richer documentation, and allows us to add special annotations, not supported by off-the-shelf RDBMS systems.

To ensure that the database schema definition is well integrated with the rest of the LSST domain model, we maintain a copy of the Baseline schema in the Enterprise Architect system.

2 Stored Procedures and Functions

We maintain a small set of stored procedures and functions, as well as a set of user-defined functions (UDFs). Since stored procedures and functions typically are vendor-specific, we have tried to limit them to the bare minimum; nevertheless, we now have ~ 10 such procedures, most related to converting different time formats. The most frequently used functions are written as native user-defined functions for performance and portability reasons. They include support for spherical geometry, photometry conversions, and statistical functions (e.g., median). These spatial functions rely on custom HTM indices to allow fast spatial selections, such as point in circle, polygon or ellipse searches. We are collaborating with the team from Johns Hopkins University that implemented HTM indexing and UDFs for SDSS, and we already incorporated many of their lessons learned into our UDFs (their UDFs were written in C# for SQL Server, since then the SDSS team converted them to C++ using automated conversion tools.)

Since these UDFs are not LSST-specific and may be of broader interest (to MySQL community and beyond), we maintain them as an open-source project independent from LSST. This project, SciSQL, is hosted on launchpad.net (see https://launchpad.net/scisql/).

3 Selected Catalogs

The baseline schema is driven primarily by the Data Products Definitions Document (LSE-163). The schema can be divided into several logical groups:

- Core Catalogs
- Metadata Catalogs
- Provenance Catalogs
- SDQA

Here, a catalog refers to a group of closely-related tables that act as a unified dataset from an end user's point of view.

3.1 Core Catalogs

The core catalogs can be divided into:

- Level 1 Catalogs: DiaObject, SSObject. DiaSource and DiaForcedSource.
- Level 2 Catalogs: Object, Source, ForcedSource.

DiaObject table contains information about astronomical objects measured on difference images. For reproducibility reasons (described in LDM-135, chapter 3.1), we expect to create a new row each time there is an update to a diaObject, thus the total number of rows in that table is higher than the unique number of diaObjects.

SSObject table contains information about Solar System (also called moving) objects.

DiaSource table contains information about individual measurements of astronomical objects on different images.

DiaForcedSource contains photometry measurements about low signal-to-noise detections done on individual difference image exposures in each place where an object was detected on a previous difference image within one month.

Object Catalog contains information about static astronomical objects measured on a stacked image. It is the most commonly used catalog.

Source Catalog contains information about high signal-to-noise (>5 sigma) detections on single frame images.

ForcedSource table contains photometry measurements about low signal-to-noise detections done on individual exposures in each place where an object was detected on a stacked image. Due to the nature of these detections only a few parameters can be measured, thus the table is very narrow, however it is by far the largest in terms of row-count.

DiaObject_To_Object_Match table keeps a mapping of diaObjects to nearby objects, used primarily for user data analysis. For rapid real-time queries we maintain three nearbyObj columns in DiaObject.

3.1.1 Sizes

The table below provides a rough count (size, rows, columns) for core tables (in the last Data Release, counting data only, e.g., size of persistent overheads such as indices is not reflected here). The exact numbers for each DR can be found in LDM-141.

Table	Size [TB]	Rows [billions]	Columns
Object (narrow)	~ 107	~ 47	~ 330
Object (all extras)	$\sim 1,200$	Largest $\sim 1,500$	$\sim 7,650$
Source	$\sim 5,000$	$\sim 9,000$	~ 50
ForcedSource	$\sim 1,900$	$\sim 50,000$	6
SSObject	~ 0.003	0.006	~ 80
DiaObject	~ 27 (76 max)	~ 15 unique (~ 44 max)	~ 260
DiaSource	~ 23	~ 45	~ 70
DiaForcedSource	~ 13	~ 300	8

We expect to keep refining the baseline schema extensively through LSST's construction, and will likely end up adding a small number of additional columns we haven't thought about so far. Since the schema is used for cost estimation, such changes would increase overall size of the database (and consequently, the cost), so we have reserved in the costing model (LDM-141) 25% of each table size[1] for yet-unknown columns. Further, the costing model assumes

[1] With the exception of ForcedSource and DiaForcedSource

a 3% increase in row size for each Data Release.

3.1.2 Partitioning

To optimize disk I/O and ensure frequently accessed columns are collocated, and to avoid unnecessary I/O, selected Catalogs are *vertically partitioned*.

The Object Catalog is partitioned vertically into the following tables:

- Object — a table with ∼330 most frequently used columns. This table is expected to contain ∼47 billion rows by the end of the survey, containing ∼48% in the first data release.
- Object_APMean — a very narrow table containing aperture photometry mean values, on average ∼8 per object.
- Object_Periodic — a very narrow table containing definitions of periodic features, on average ∼32 per object.
- Object_NonPeriodic — a very narrow table containing definitions of non-periodic features, on average ∼20 per object.
- Object_Extra — a very wide table with remaining, less frequently used columns, such as covariances for 2 models (Point Source and Bulge+Disk), and Bulge+Disk samples. These are packed into 3 blobs, 66, 168, and 7,200 elements in each blob respectively. Packing is driven by expected access pattern: entire set of elements in a blob or nothing will be accessed. There will be one such row for each row in the Object table.

The Source Catalog consists of two tables: Source and Source_APMean, the latter is similar to Object_APMean.

Since managing trillions of rows in a single table is close to impossible, in addition to vertical partitioning, all core catalogs will be horizontally partitioned, The Level 1 catalogs will be partitioned by time into a small set of tables, and all large Level 2 catalogs will be partitioned spatially into ∼20,000 partitions for each table. Further details related to horizontal partitioning of these tables are discussed in LDM-135 (*LSST Database Design*).

3.1.3 Diagram

The diagram below depicts all core tables, and their relations.

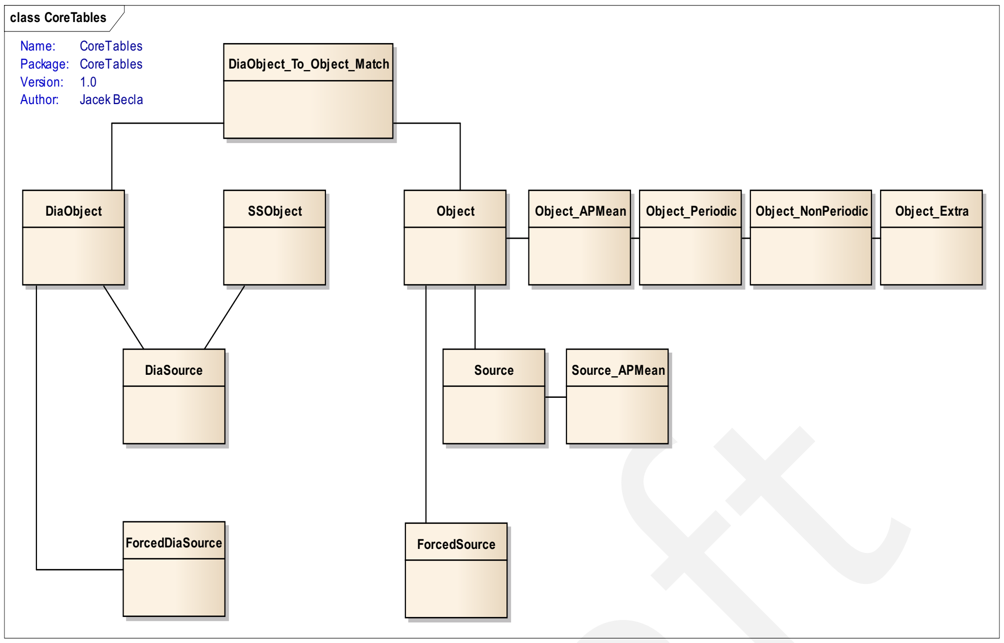


FIGURE 1: Core table relationships

3.2 Metadata Catalogs

Metadata catalogs contains information about LSST “raw exposures” and “science calibrated visits”. Information is tracked at different levels:

- focal plane (entire exposure/visit),
- raft (raft = 9 ccds)
- ccd (there are 189 ccds per focal plane)
- amp (there are 16 amplifiers per ccd).

When appropriate, we denormalize information into “lower-level” table (eg, information from ccd-level is repeated for each amp) to avoid extraneous joins for common accesses.

In addition to fully structured tables, for selected tables we expect to maintain an additional table with key-value pairs. This will allow us to introduce additional “columns” without altering corresponding table’s schema (at the cost of degraded performance). This might be particularly handy when trying to determine what information would be particularly useful to derive from the existing columns. Periodically, most commonly used key-value pairs will be converted into regular columns.

In addition, we maintain a table that maps raw exposures to visits. Typically it will contain two rows for each visit.

3.2.1 Sizes

Size of the metadata tables is not posing any major challenge. LSST is expected to produce ~one thousand visits per night, which leads to ~3 million visits during 10 year lifetime of the survey.

3.2.2 Diagram

The diagram below depicts the metadata tables, and their relations.

3.3 Provenance

In LSST, we define Provenance as execution trace, or in other words information "how we came up with the data". Provenance will be used primarily:

- to help with data QA, for example to detect which parts of data were affected by a faulty algorithm or a bad node,
- as a recipe how to regenerate intermediate results.

Specifically, Provenance is responsible for capturing configuration of:

- software (which pipelines were run on given data, what processing stages were part of each pipeline, what versions of algorithms were used, how was processing parallelized and on which computing node each piece was executed etc) and
- hardware (what was the configuration of focal plane, each raft, ccd, amplifier, filter etc).

3.3.1 Size

In terms of size, we expect provenance to be a very small fraction of the entire data set. The largest contribution is an 8-byte column we are adding for every core tables (with the ex-

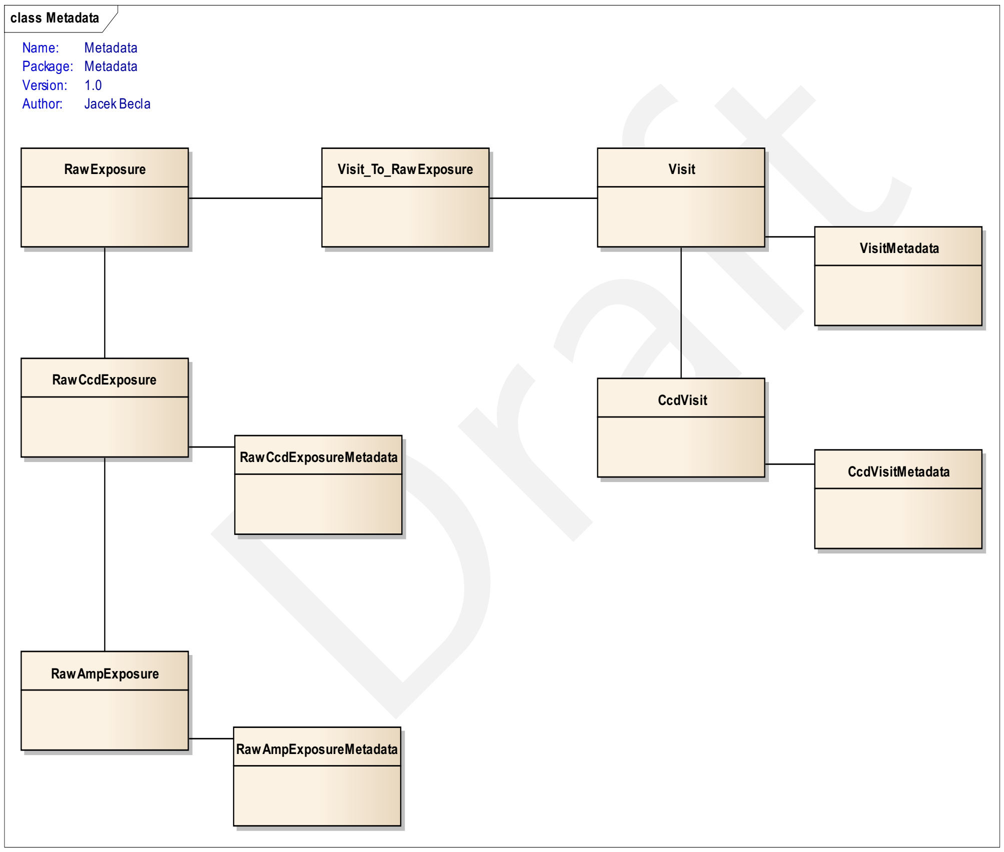
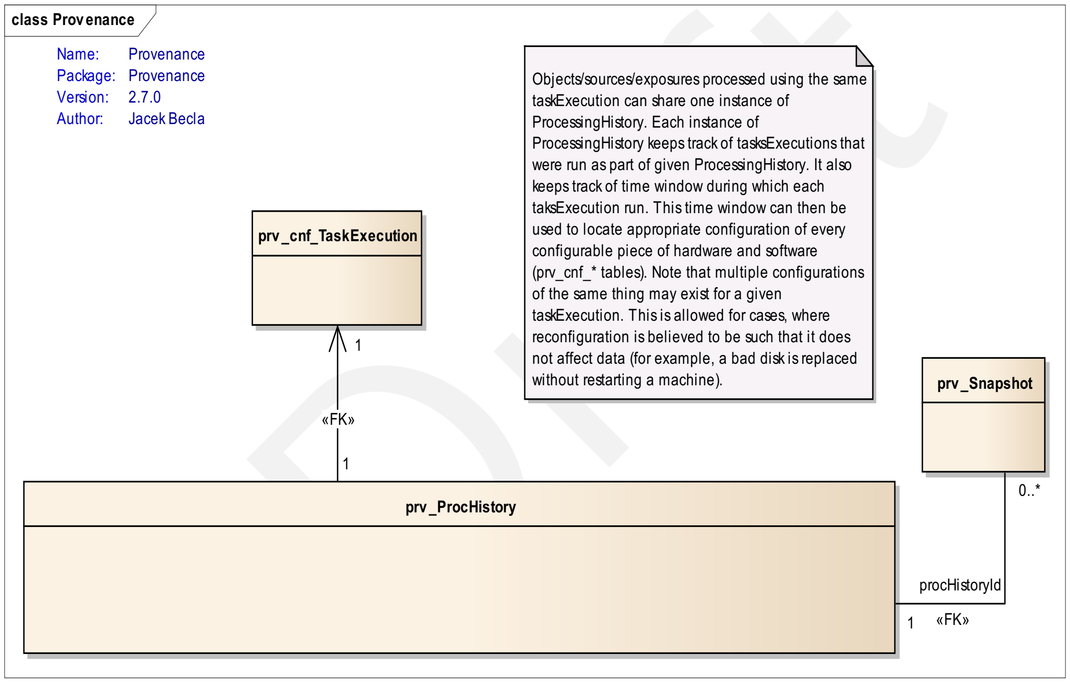


FIGURE 2: Metadata table relationships

ception of DiaForcedSource table, as it will have the same provenance as its corresponding DiaSource table), and selected metadata tables. Note that in some cases, in particular for the very narrow ForcedSource table with trillions of rows, that contribution is non-negligible.

3.3.2 Diagrams

The diagram below depicts the metadata tables, and their relations. Note that there is a separate section discussing provenance design in details, see below *Selected Design Aspects* below.



Objects/sources/exposures processed using the same taskExecution can share one instance of ProcessingHistory. Each instance of ProcessingHistory keeps track of tasksExecutions that were run as part of given ProcessingHistory. It also keeps track of time window during which each taksExecution run. This time window can then be used to locate appropriate configuration of every configurable piece of hardware and software (prv_cnf_* tables). Note that multiple configurations of the same thing may exist for a given taskExecution. This is allowed for cases, where reconfiguration is believed to be such that it does not affect data (for example, a bad disk is replaced without restarting a machine).

FIGURE 3: Provenance table relationships

3.4 Other Tables

In addition to the catalogs/tables described above, there are several groups of other tables in the baseline schema. In terms of size and complexity they pose very little challenge comparing to the catalogs described above, thus this document does not focus on discussing these tables (and we have left designing details of these tables for closer-to-commissioning). They include Software Data quality Assurance tables (prefixed with sdqa_), used for tracking various quality aspects of the LSST catalogs.

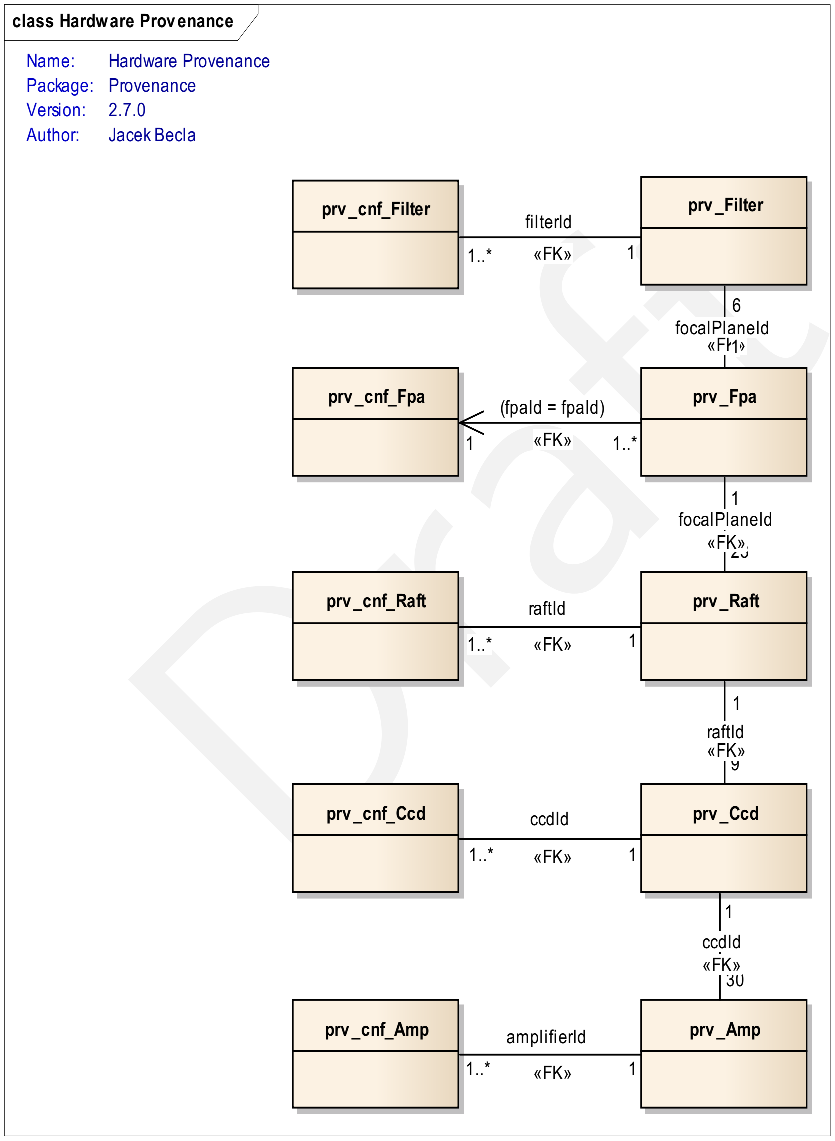


FIGURE 4: Hardware provenance table relationships

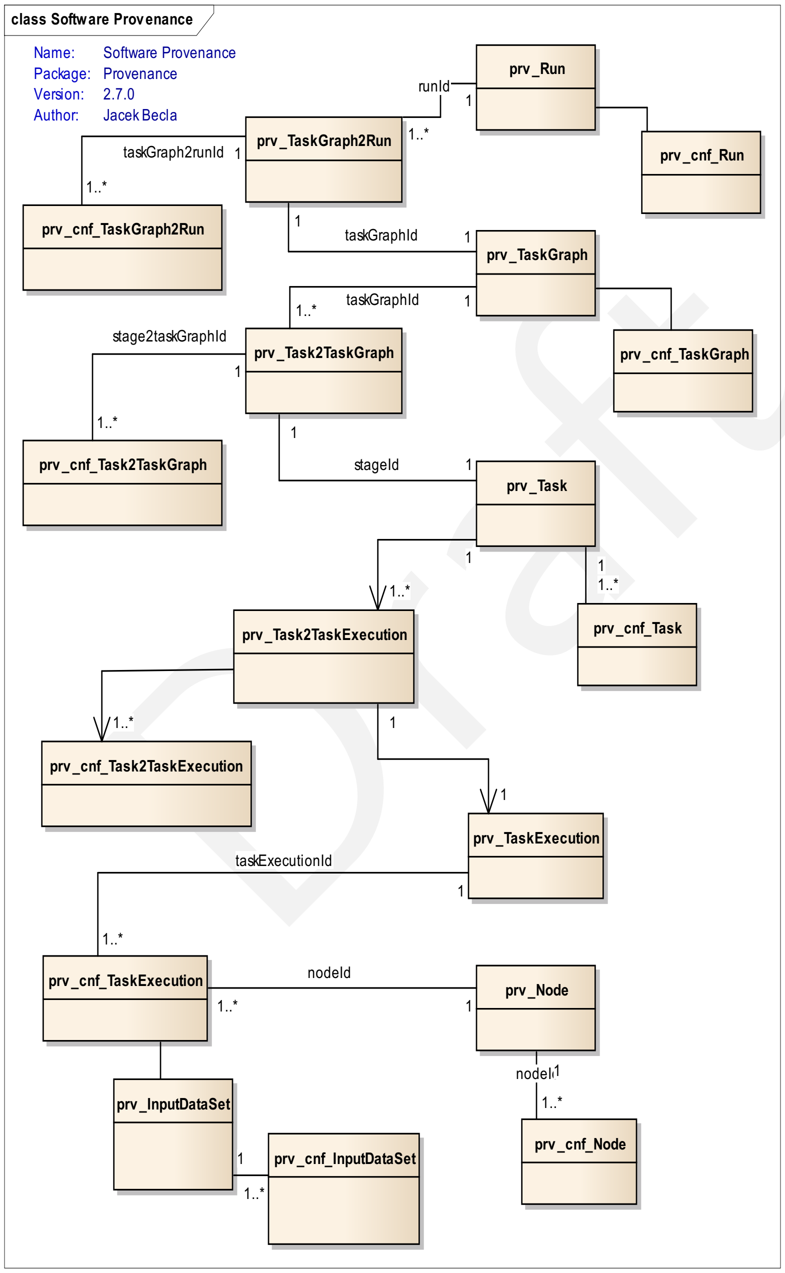


FIGURE 5: Software provenance table relationships

4 Selected Design Aspects

Below we covered in details selected, most interesting aspects of the schema design.

4.1 Provenance

Provenance is fundamentally concerned with two things: recording the state of relevant portions of the system and associating state with the generation of an output from inputs. Given the size of data, provenance information cannot be captured per data element. Instead data elements with the same provenance should be grouped. That in turn is non-trivial due to the way data elements will be built and updated:

- Any given data element will be the result of the processing of raw data by a Task Graph (also called a pipeline). Each Graph consists of many Tasks. Each task may involve executing multiple algorithms.
- It is anticipated we will need to fix bad data (as long as it is unreleased) by reprocessing it (e.g. using a fixed algorithm), so in practice different parts of the same data element might have many different processing histories.
- Finally, there is a large number of dependencies. Here is just a small example: object data depends on a coadd, which is generated using masks derived from difference sources which depend on the corresponding difference image which depends on a template image which depends on calibrated science images and so on...

Also, provenance should be flexible enough to easily allow adding new items to the system state.

For the above reasons, simple solutions like timestamping every data element will not work.

In practice, we plan to implement the state recording function of Provenance using time ranges, with granularity up to 1 sec. Every configurable piece of information will be tracked through two tables: one that keeps definitions and the other that keeps configurations. For example, each filter (u, g, r, i, z, y) will be represented as one row in prv_Filter table (there will be exactly 6 definitions), and each configuration of each filter will be represented as one row in prv_cnf_Filter table (initially there will be 6 configurations: one for each filter, but if we happen to later break one filter and will introduce a new one, a new configuration will need to be inserted). Each configuration will have a validity time range assigned (time period when it was valid).

The heart of the processing recording function of Provenance is the *prv_ProcHistory* table; it has two responsibilities:

- Assign a unique processing history id (procHistoryId) for each creation of a set of output data elements. An 8-byte integer is sufficient to provide the needed granularity.
- Bind each unique procHistoryId with a set of executions of tasks, including the inputs to those tasks, identified by their own procHistoryIds. (A task is the smallest chunk that will be executed atomically, that is, if part of a task fails, the whole task will be rolled back and re-run). Typically there will be one task execution for each procHistoryId, but in cases of reprocessing there may be more than one.

Notice that procHistoryId is not associated with a time range, but with a set of task executions, each having its own configuration, its own time range, and its own node it runs on.

The Provenance may also track which columns for each table are updated by each stage (we require that each column is updated by a single stage only - for that reasons some columns, eg. flags had to be split into multiple columns).

Every data element that needs to have provenance tracked (for example, every row in the Object, Source, or Exposure table) will need to store a procHistoryId. Note that a single row might be generated or updated by different pipelines or task graphs, and each may run at different time, on different hardware, possibly even at a different site. Having a procHistoryId will allow it to find which task graphs and tasks were executed for it, at what time, and on which processing nodes. It will then be possible to correlate the time periods when each task was executed with the configurations that were valid at those times.

Such an approach has two important pros:

- **Flexibility**: It allows us to decouple configurable elements from Objects, Sources, Exposures, etc. — they are very loosely coupled through time range only. In particular, it means that new configurable items can be added or removed from the system at any given time with no need to do schema evolution and no need to update existing data.

- **Space-efficiency**: It is clear that components of the system will be changing relatively frequently. Minimizing the storage required to track these changes is essential. In our case, a new configuration will only require storage for that configuration. Each Object/Source/Exposure only needs to store one procHistoryId to get access to hundreds of different configurations.

We expect we will need to tune the current approach to provide efficient query access to the Provenance. In particular, the implementation relies on range queries which may need special optimizations, or special indexes.

4.2 Schema Evolution

In summary, we expect to avoid heavy changes that alter shape of large tables (e.g. by preallocating space), and rely on partitioning to scale and run necessary schema changes fast and scalably in parallel[2].

4.2.1 Released Data Sets

Each data release is independent, therefore the database schema can easily change between data releases; we do not anticipate any non-trivial challenges here.

Adding new columns. We are planning to keep a few extra unused columns of each common type (for example, 3 FLOATs, 3 DOUBLEs, 3 INTs, 3 BIGINTs) for each large table, and use one of them when a new column is needed. (renaming a column is a trivial and instantaneous operation). Speed of filling new columns should be comparable to speed of a full table scan.

Updating existing columns. Speed of updating values should be comparable to speed of a single shared scan.

Deleting columns. If we have to delete a column, instead of deleting it, (which is expensive as it changes shape of a table), we will add it to the pool of ”extra, unused columns” by renaming it.

The unsed/hidden columns will be hidden through non-materialized views. It is likely that we will end up providing multiple views, e.g. each time we make a schema change, we'd expose the changes through a new view.

[2]For more information about the schema evolution tests we run, refer to https://dev.lsstcorp.org/trac/wiki/db/tests/SchemaEvolution

Non-trivial changes. Should we ever need to do non-trivial changes: Non trivial schema changes on multi-billion row table tend to take extremely long time (measured in days). In our case, each large table will be partitioned into several thousand chunks, which makes the updates scalable and faster: not only the updates can be done in parallel on multiple machines, but each partition is small enough to be handled efficiently without running into any bottlenecks or problems with lack of buffer space.

Notes on administration. All queries accessing the database will go through our interface, which intercepts and interprets the queries. This will allow us to put the entire system in special mode (user queries will be queued, administrative queries will continue to work). The administrative tools we are building will allow us to push the necessary updates to all nodes, and we could use the existing scheduling mechanisms to run the upgrade in the most efficient fashion. The admin tools will also allow running necessary verifications (e.g., whether the schema for all table chunks matches).

4.2.2 Level 1 (Up-to-date Catalog)

We will have opportunity to update schema every time we are replacing Level 1 catalog after each data release. Should we need to make the updates in between, the changes can easily be done during the time when we are not taking data (daytime).

5 Naming Conventions

We have defined a set of naming convention for naming tables, columns and relationships. In summary, the format of table names follows:

```
<_><prefix><purpose>_<extend>_<component><postfix>
```

Leading _ is used to show it is an internal table.

Prefix:

- All Provenance tables are prefixed with prv_, example: prv_Amp. Further, it may be followed by cnf_ which is used for provenance tables containing individual configurations. For example, one row in prv_Ccd contains information about one particular Ccd, and prv_cnf_Ccd contains information about one specific configuration version of one particular Ccd.

Purpose: it includes things like Raw, Science, Calibration, Bias, Dark, Flat, Fringe, IR. (we have recently dropped "Science" for *Visit tables).

Extend: Fpa, Ccd, Amp (we have dropped "Fpa" from raw and science table names).

Component: Exposure, Object, Source, Psf, Wcs...

Postfix: optional

Mapping tables follows its own naming scheme: _<table X name>_To_<table Y name>

6 Baseline Schema

A References

References