FASTDB Technical Overview

Rob Knop

DESC FASTDB Workshop

2025 July

FASTDB Technical Design Overview

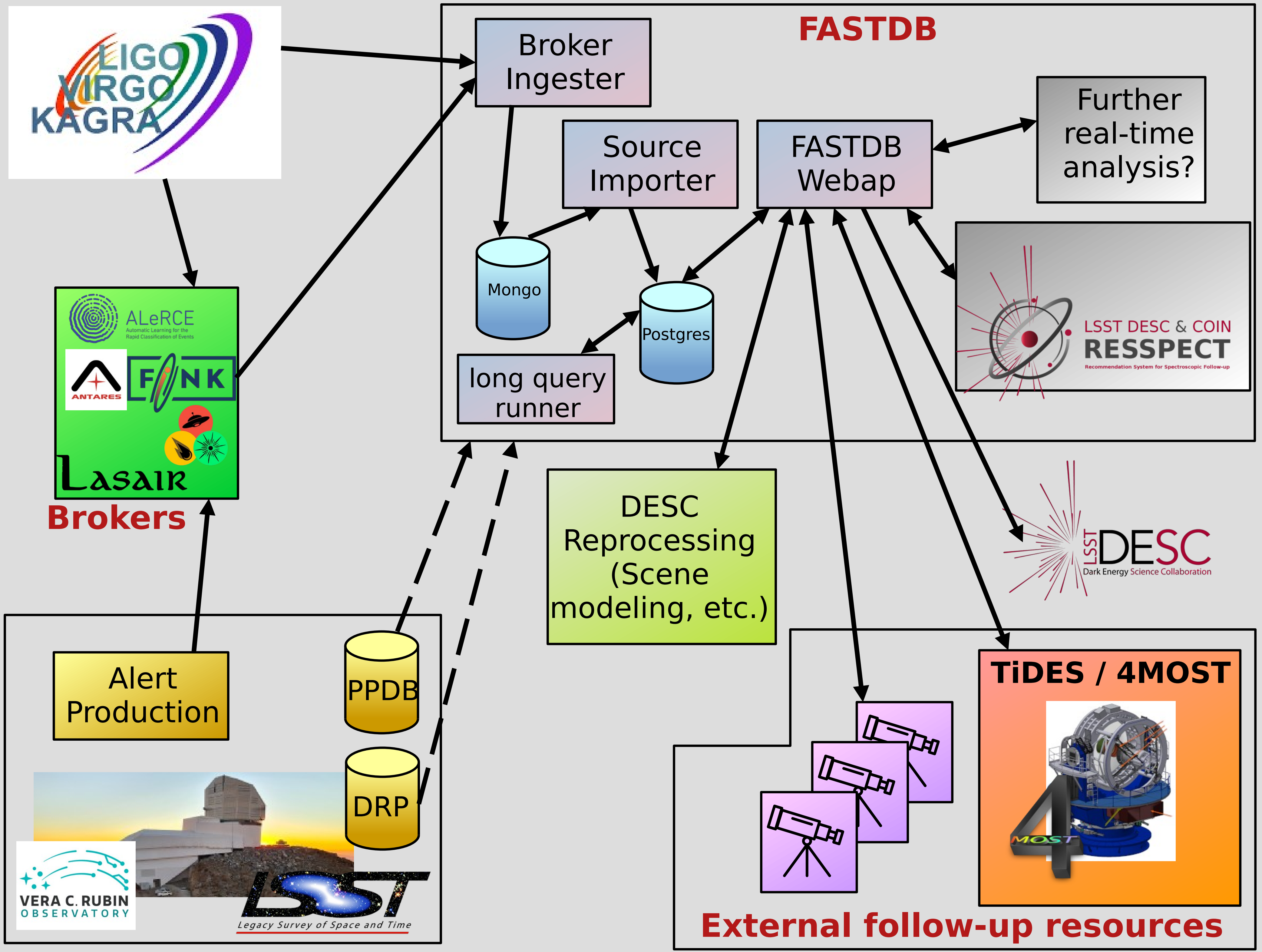
- Runs as a set of separate services in containers (launched via kubernetes, or, for tests, docker).
- Primary instance runs on NERSC Spin. (There are thus far only musings for backup / failover / replication instances.)
- Defined by PostgreSQL schema, but also by a set of API interfaces and realtime services, and by a broker message format (stored, at least temporarily, in MongoDB).
- Minimal use of frameworks— no ORM framework (those are cumbersome), lightweight web app framework (flask).
- Database migrations written directly in SQL, applied sequentially and tracked by a lightweight custom migration manager.
- <neo> Tests. Lots of tests. </neo>

FASTDB Technical Design Overview (continued)

- Primary storage is in a PostgreSQL database.
- Secondary storage is in a MongoDB database.
- Realtime services for monitoring broker alert streams.
- External access is via a web API front-end, using either a provided client or just hitting the API directly.
- RESSPECT integration.
- Most of DESC has read-only access.
- “Processing Version” and “Snapshot” for storing multiple versions of the same objects’ lightcurves.

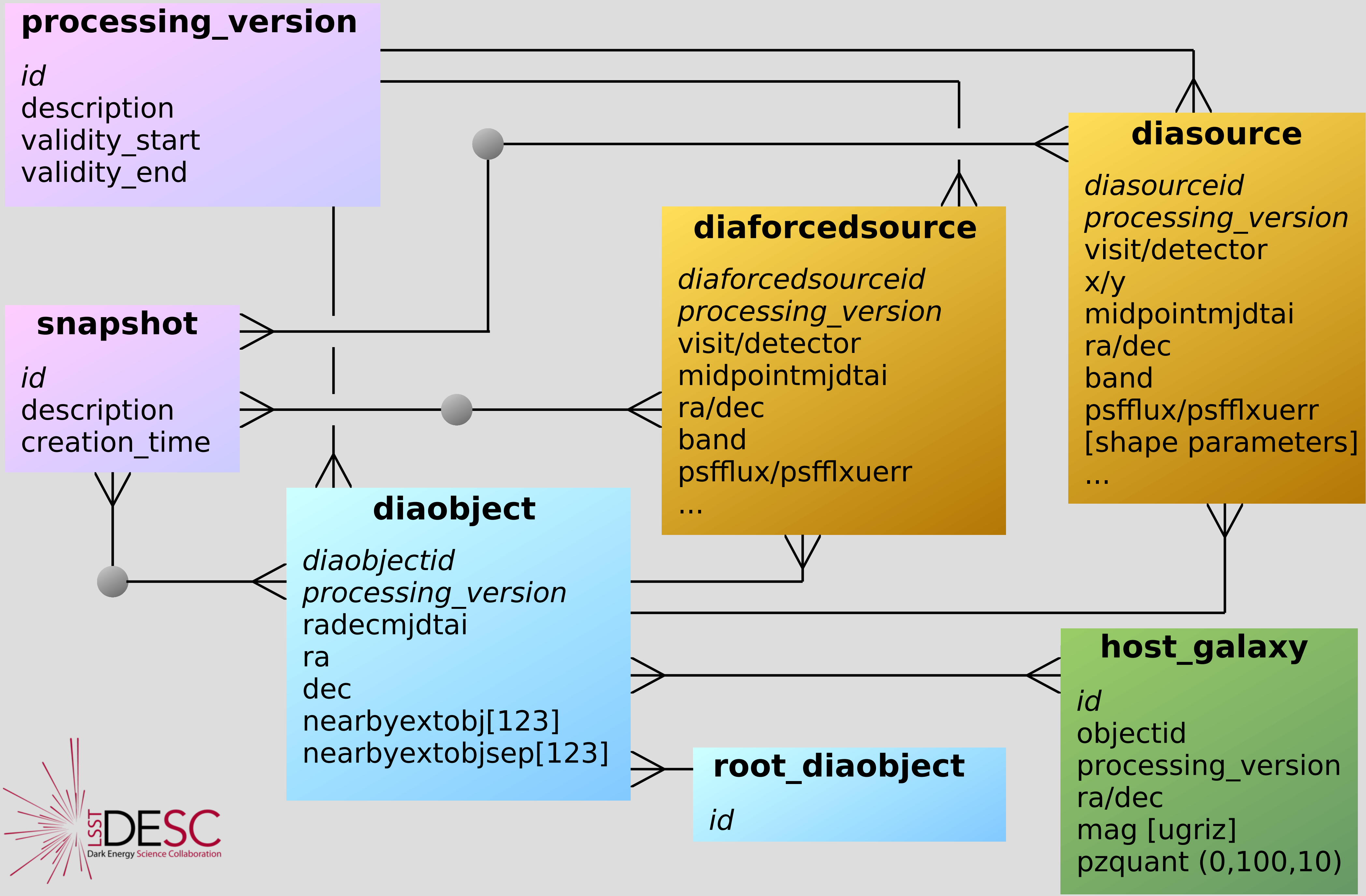
What FASTDB is not

- Just an SQL database. (You can't access the database directly — but see several slides ahead.)
- A full TOM. (If we want observatory submission functionality, we should export specific objects to a TOM such as SkyPortal.)
- A place for users to track and comment on their favorite objects. (Again: SkyPortal exports.)

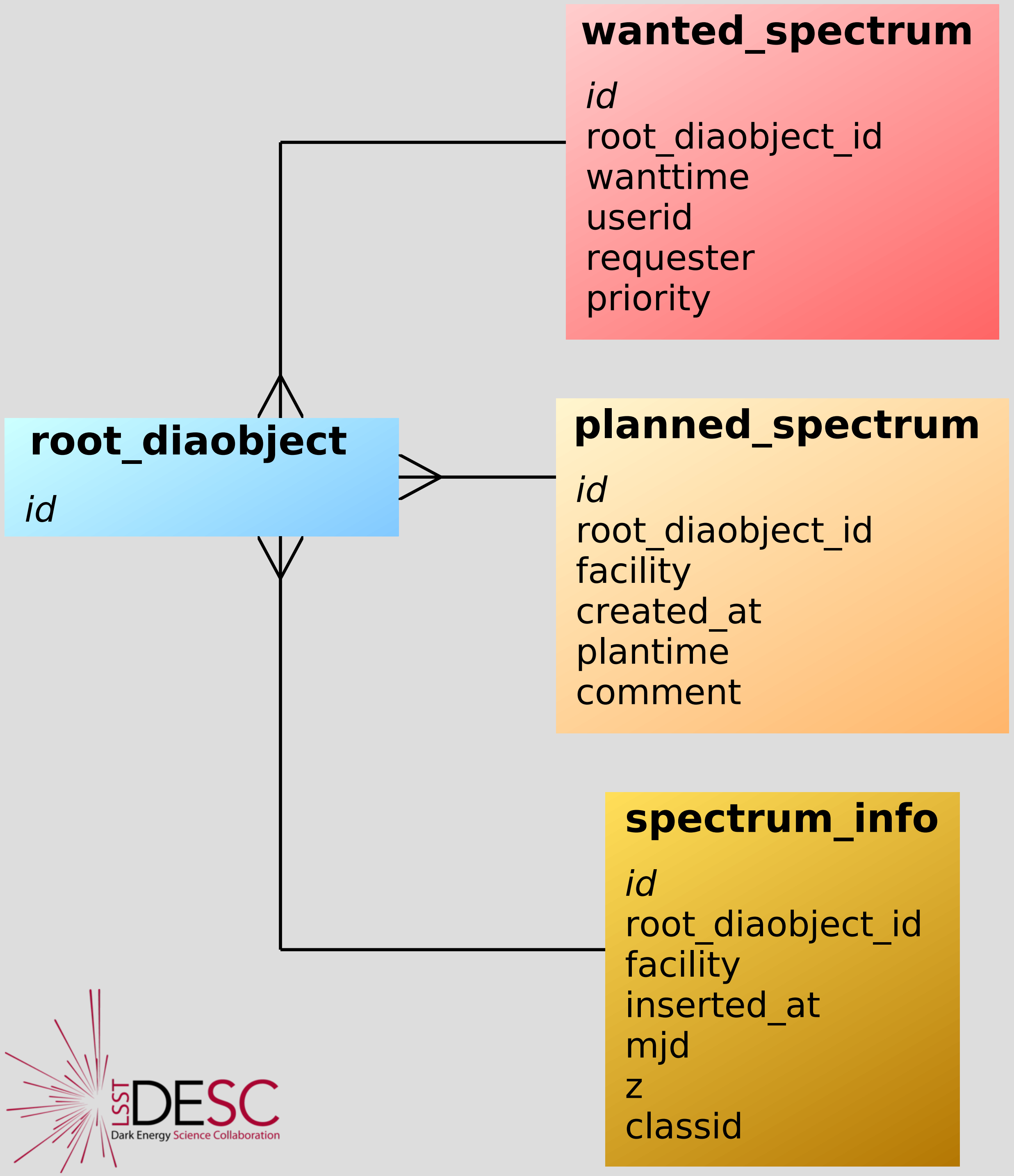

FASTDB
Broker Ingester
Source Importer
FASTDB Webap
Further real-time analysis?
Mongo
Postgres
long query runner
LSST DESC & COIN RESSPECT
Recommendation System for Spectroscopic Follow-up
LIGO VIRGO KAGRA
ALeRCE
Automatic Learning for the Rapid Classification of Events
ANTARES
FINK
Lasair
Brokers
DESC Reprocessing (Scene modeling, etc.)
LSST DESC
Dark Energy Science Collaboration
Alert Production
PPDB
DRP
VERA C. RUBIN OBSERVATORY
LSST
Legacy Survey of Space and Time
TiDES / 4MOST
4MOST
External follow-up resources

FASTDB Lightcurve Tables

Based on APDB and DPDD 4.3.1



FASTDB Spectrum Cycle Tables



Designed for interaction wtih RESSPECT.

Note: FASTDB implementation is not complete.

TODO: storage of actual spectra: what information? format? location?

TODO: distinguish host and transient spectra

Processing Version

- Lightcurve tables have a `processing_version` column
- This is part of the table's primary key!
- One consequence is that you have to know what you're doing when constructing SQL queries; the obvious thing may be wrong!
- The processing_version_alias table allows us to assign multiple names to the same processing version.
- Anticipate having a "default" alias that points to what people should get if they don't know what to get.

diaforcedsource

diaforcedsourceid
processing_version
visit/detector
midpointmjdtai
ra/dec
band
psfflux/psfflxuerr
...

processing_version

id
description
validity_start
validity_end

processing_version_alias

id
description

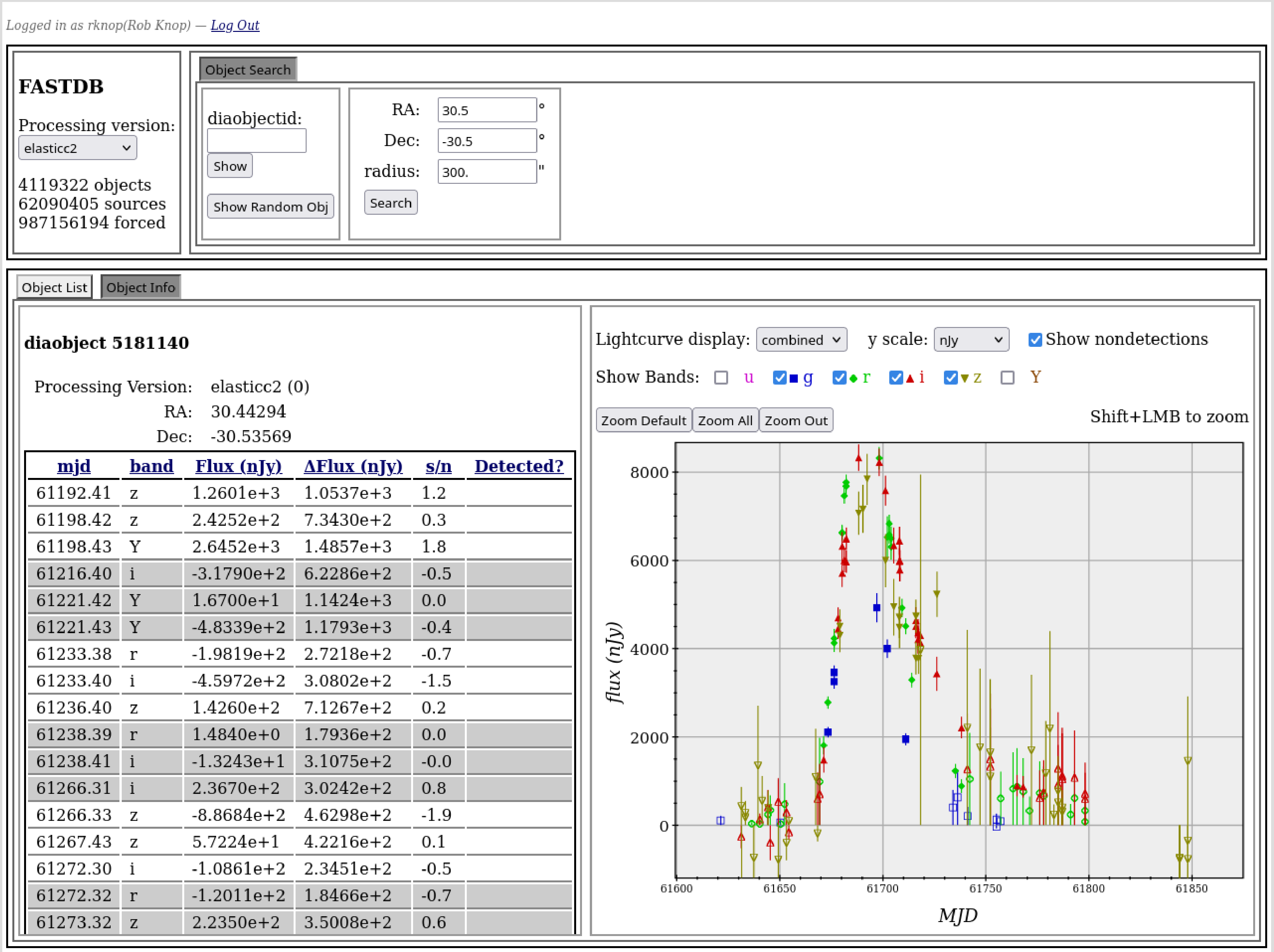
Aside: comment from the LS4 survey paper

L273-275 "A unique aspect of SeeChange is that it is designed to _be able_ to run the same data"...

Overall, this paragraph seemed a bit aspirational. I'm familiar with enough astronomical projects that said they were going to do this provenance tracking, but didn't achieve this goal; and certainly never in a way that this information was actually ever use to re-run something for science comparison. If this a design plan, then great. If this a capability that already exists, as suggested by this section, could you add a short table that gives an example of tracking different versions for different data products? That would make me take this more seriously.

(The LS4 pipeline *does* already have extensive provenance tracking [more advanced than what FASTDB has, or what would be appropriate for FASTDB].)

FASTDB Access — Web UI



Logged in as rknop(Rob Knop) — Log Out

FASTDB

Processing version: elasticc2

4119322 objects
62090405 sources
987156194 forced

Object Search

diaobjectid:

Show

Show Random Obj

RA: 30.5 °
Dec: -30.5 °
radius: 300. "

Search

Object List | Object Info

diaobject 5181140

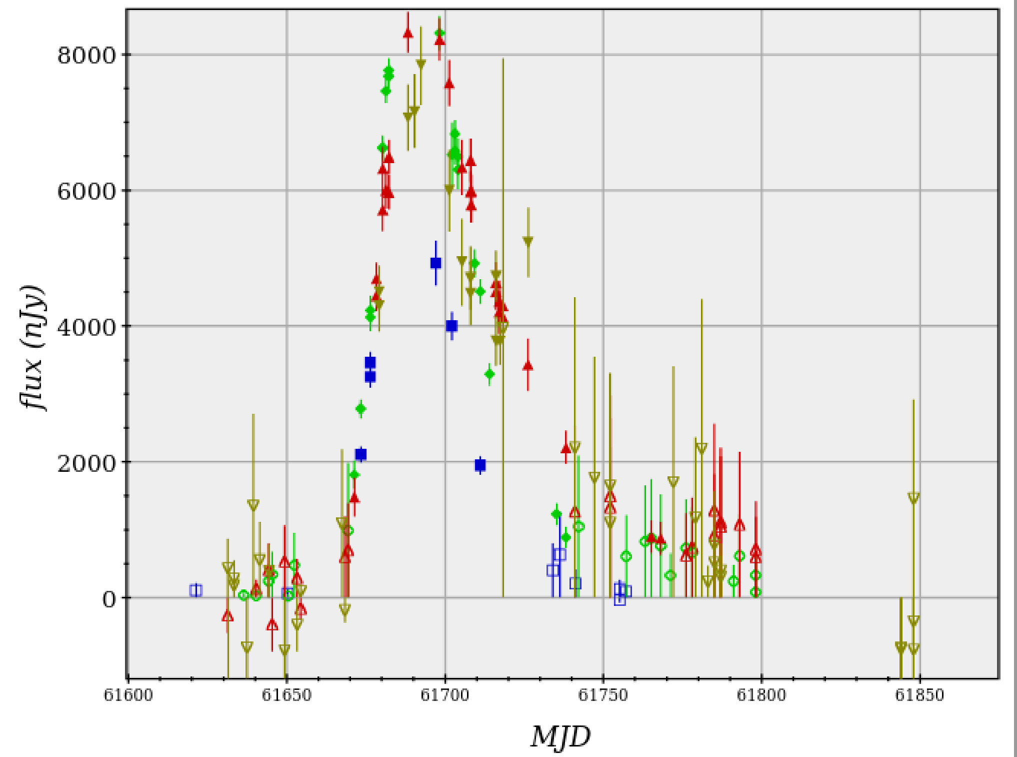
Processing Version:	elasticc2 (0)
RA:	30.44294
Dec:	-30.53569

mjd	band	Flux (nJy)	ΔFlux (nJy)	s/n	Detected?
61192.41	z	1.2601e+3	1.0537e+3	1.2	
61198.42	z	2.4252e+2	7.3430e+2	0.3	
61198.43	Y	2.6452e+3	1.4857e+3	1.8	
61216.40	i	-3.1790e+2	6.2286e+2	-0.5	
61221.42	Y	1.6700e+1	1.1424e+3	0.0	
61221.43	Y	-4.8339e+2	1.1793e+3	-0.4	
61233.38	r	-1.9819e+2	2.7218e+2	-0.7	
61233.40	i	-4.5972e+2	3.0802e+2	-1.5	
61236.40	z	1.4260e+2	7.1267e+2	0.2	
61238.39	r	1.4840e+0	1.7936e+2	0.0	
61238.41	i	-1.3243e+1	3.1075e+2	-0.0	
61266.31	i	2.3670e+2	3.0242e+2	0.8	
61266.33	z	-8.8684e+2	4.6298e+2	-1.9	
61267.43	z	5.7224e+1	4.2216e+2	0.1	
61272.30	i	-1.0861e+2	2.3451e+2	-0.5	
61272.32	r	-1.2011e+2	1.8466e+2	-0.7	
61273.32	z	2.2350e+2	3.5008e+2	0.6	

Lightcurve display: combined   y scale: nJy   ☑ Show nondetections

Show Bands: ☐ u ☑ ■ g ☑ ◆ r ☑ ▲ i ☑ ▼ z ☐ Y

Zoom Default | Zoom All | Zoom Out   Shift+LMB to zoom



FASTDB Access — fastdb_client.py

Should work with the the **desc-td-dev** environment.
(as pf monday; thank you Heather!)

```
source $CFS/lsst/groups/TD/setup_td_dev.sh
```

To use: put `fastdb_client.py` somewhere in your PYTHONPATH, or (on NERSC) do:

```
import sys
sys.path.insert(0, '/global/cfs/cdirs/desc-td/SOFTWARE/'
                   'fastdb_deployment/fastdb_client' )
```

Then run:

```
from fastdb import FASTDBClient
fastdb = FASTDBClient( server, username, password )
```

where server is the URL of the FASTDB web ap, and username and password are your account on that server. (Easier way: see next page.)

FASTDB Access — fastdb_client.py

You can make life easier on yourself by creating a file `~/.fastdb.ini` with contents:

```
[rknop-dev]
url = https://fastdb-rknop-dev.lbl.gov
username = <your username here>
password = <your password here>
```

You can have multiple different servers defined in blocks each starting with a name in brackets.

After creating this file, do

```
chmod go-rwx ~/.fastdb.ini
```

to make sure that others can’t read your password.

Now you can connect to FASTDB by putting the thing in brackets in place of the server:

```
fastdb = FASTDBClient( ‘rknop-dev’ )
```

FASTDB Access — fastdb_client.py

A `FASTDBClient` behaves (sort of) like a thin front-end to python requests. You can post to a (relative!) web URL, pass data in a dictionary (if the web URL is expecting that), and get the response (with JSON parsed to a python object):

```
>>> res = fastdb.post( '/getprocvers' )
>>> print(res)
{'status': 'ok', 'procvers': ['elasticc2']}
```

Why use FASTDBClient instead of just python requests?

- Handles authentication handshaking (painful!)
- Relative URLs
- Built-in retry loop

FASTDB Web API — processing versions

...ideally, all the API endpoints will be documented in the FASTDB documentation. That is still very much in progress.

`/getprocvers` — returns a dictionary; in `'procvers'` is a list of all known processing versions and aliases (strings).

`/getprocver/<pv>` — looks up the processing version defined by either numeric id, description, or alas <pv>. Returns a dictionary with keys `'id'` (integer id), `'description'` (name of processing version), and `'aliases'` (list of strings).

FASTDB Web API — object search

`/objectsearch/<pv>` — Searches for objects in the processing version (or alias) <pv>

- Pass it a JSON-encoded dictionary as the POST body. (You can specify the dictionary to FASTDBClient.post with the json= parameter.)

- Currently supported (...and required...) parameters in the POST body:
  `ra` — decimal degrees or HH:MM:SS
  dec — decimal degrees or dd:mm:ss
  `radius` — arcseconds

  (Aside: detected, forced, "patch" lightcurves.)

- Anticipate adding lots of parameters to expand search capability. (CF: DECAT-DDF webap.)

FASTDB Web API — lightcurves

`/ltcv/getltcv/<pv>/<objid>` — return a dictionary with lightcurve information for object `<objid>` in processing version (or alias) `<pv>`.

Returned dictionary has two keys, `objectinfo` and `ltcv`. The value associated with `'ltcv'` is itself a dictionary, with keys `mjd`, `band`, `psfflux`, `psffluxerr`, `isdet`, (`ispatch`)

`/ltcv/getrandomltcg/<pv>` — Same return as `getltcv`, only picks a random diaobject from processing version <pv>.

`/ltcv/gethottransients` — Get lightcurves of current active transients. Parameters specifying the search passed as a JSON-encoded dict. More info (ideally) in the FASTDB documentation.

FASTDB Web API — the spectrum cycle

`/spectrum/askforspectrum` — indicate a that a given object is a priority for obtaining spectroscopic follow-up.

`/spectrum/spectrawanted` — get a list of priority objects for spectroscopic follow-up.

`/spectrum/planspectrum` — indicate the intention to take a spectrum of an object.

`/spectrum/removespectrumplan` — remove previously submitted plan (e.g. weather)

`/spectrum/reportspectruminfo` — report spectrum, z, classification

`/spectrum/getknownspectruminfo` — get which objects have spectra, z, classifications

All of these take parameters as POST data as a JSON-encoded dict, return JSON with information (or at least a status).

FASTDB API — direct SQL queries

You can send via web API a series of SQL queries that will be executed sequentially by the SQL server. (The series is so that you can do things like create temporary tables.) You will get the output from the last query in the series.

Two interfaces:

“Short queries”: queries that will complete in less than 5 minutes. (You can submit longer queries, but the web app will time out before the database finishes the query, so you’ll never get the response.)

“Long queries” : you submit a query with one API endpoint, and then hit other API endpoints to check if it’s done and get results. fastdb_client.py has a simplified interface to this.

FASTDB API — direct SQL queries

Reasons to discourage use:

- Easy to make incorrect queries. (Do you fully understand the schema WRT processing version?) For things that will be commonly used, we should make API endpoints.
- Easy to write inefficient queries. (Postgres sometimes is stupid and requires hand-tuning of query plan hints.)

FASTDB API — direct SQL queries

`fastdb_client.FASTDBClient` methods:

```
submit_short_sql_query(query, subdict=None,
  return_format='csv')

synchronous_long_sql_query( query,
  subdict=None, return_format='csv',
  checkeach=300, maxwait=300)

submit_long_sql_query( query, subdict=None,
  return_format='csv')

check_long_sql_query(queryid)

get_long_sql_query_result(queryid)
```

https://github.com/LSSTDESC/FASTDB

`admin` — yaml files defining spin deployment

`client` — fastdb_client.py

`db` — PostgreSQL database migrations

`docker` — defines the docker imgae used on Spin and for tests

`docs` — sphinx docs

`examples` — (thus far) one jupyter notebook with examples

`notes` — notes I've made to myself

`share` — source locations of .avsc files defining alert schema

`src` — source code for the web server, other services, admin utilities, and the libraries used by the rest

`tests` — pytest tests

More this afternoon!