

# Description

These tests are aimed at determining how to partition the Object and DIASource tables to support efficient operation of the Association Pipeline (AP). The task of the AP is to take new *DIA sources* produced by the Detection Pipeline (DP), and compare them with everything LSST knows about the sky at that point. This comparison will be used to generate *alerts* that LSST and other observatories can use for followup observations, and is also used to bring LSST's knowledge of the sky up to date.

The current AP design splits processing of a Field-of-View (FOV) into 3 phases. For context, here is a brief summary:

## prepare phase

This phase of the AP is in charge of loading information about the sky that falls within (or is in close proximity to) a FOV into memory. We will know the location of a FOV roughly 30 seconds in advance of actual observation, and this phase of the AP will start when this information becomes available. The Object, DIASource, and Alert tables contain the information we will actually be comparing new *DIA sources* against. Of these, Object is the largest, DIASource starts out small but becomes more and more significant towards the end of a release cycle, and Alert is relatively trivial in size.

## compare-and-update phase

This phase takes new *DIASources* (produced by the DP) and performs a distance based match against the contents of Object and DIASource. The results of the match are then used to retrieve historical *alerts* for any matched *objects*. The results of all these matches and joins are sent out to compute nodes for processing - these compute nodes decide which *objects* must be changed (or possibly removed), which *DIA sources* correspond to previously unknown *objects*, and which of them are cause for sending out an *alert*. At this point, the AP enters its final phase.

## post-processing

The responsibility of this phase is to make sure that changes to Object (inserts, updates, possibly deletes), DIASource (inserts only), and Alert (inserts only) are present on disk. There is some (TODO: what is this number) amount of time during which we are guaranteed not to revisit the same FOV.

Note that LSST has a requirement to send out alerts within 60 seconds of image capture (there is a stretch goal of 30 seconds). Of the 3 AP phases, only **compare-and-update** is in the critical timing path. The telescope will be taking data for 1 FOV every 37 seconds: 15 sec exposure, 2 sec readout, 15 sec exposure, 5 sec readout and slew.

This is illustrated by the following (greatly simplified) diagram:

In this diagram, processing of a FOV starts right after observation with the image processing pipeline (IPP) which is followed by the DP, and finally the AP. The yellow and red boxes together represent processing which must happen in the 60 second window. Please note the boxes are not drawn to scale - IPP and DP are likely to take up more of the 60 second window than the diagram suggests. Also note that interaction with the moving object pipeline (MOPS) is omitted, but that there is some planned interaction between AP and MOPS (notably when a *DIA source* is mis-classified as an *object* rather than a *moving object*).

The database tests are currently focused on how to partition Object and DIASource such that the **prepare phase** is as fast as possible, and on how to perform the distance based crossmatch of the **compare-and-update phase**. Tests of database updates, inserts, and of how quickly such changes can be moved from in-memory tables to disk based tables will follow at a later date.

The tests are currently being performed using the USNO-B catalog as a standin for the Object table. USNO-B contains 1045175763 objects, so satisfies the DC2 requirement of simulating LSST operations at 10% scale for DR1 (10 billion objects).

## Code

The code for the tests [is available here?](#). The following files are included:

lsstpart/Makefile	Builds objgen, chunkgen, and bustcache
lsstpart/bustcache.cpp	Small program that tries to flush OS file system cache
lsstpart/chunkgen.cpp	Generates chunk and stripe descriptions
lsstpart/crossmatch.sql	SQL implementation of distance based crossmatch using the <u><a href="#">?zone algorithm</a></u>
lsstpart/distribution.sql	SQL commands to calculate the spatial distribution of the Object table (used to pick test regions and to generate fake <i>Objects</i> )
lsstpart/genFatObjectFiles.py	Generates table schemas for "fat" tables
lsstpart/objgen.cpp	Program for generating random (optionally perturbed) subsets of tables, as well as random positions according to a given spatial distribution
lsstpart/prepare.bash	Loads USNO-B data into Object table
lsstpart/prepare_chunks.bash	Creates coarse chunk tables from Object table
lsstpart/prepare_diasource.bash	Uses objgen to pick random subsets of Object in the test regions (with perturbed positions). These are used to populate a fake DIASource table
lsstpart/prepare_fine_chunks.bash	Loads fine chunk tables from Object
lsstpart/prepare_stripes.bash	Loads stripe tables (indexed and clustered on <i>ra</i> ) for the test regions from Object
lsstpart/prepare_zones.bash	Clusters stripe tables generated by prepare_stripes.bash and clusters them on ( <i>zoneId, ra</i> )
lsstpart/prng.cpp	Pseudo random number generator implementation
lsstpart/prng.h	Pseudo random number generator header file
lsstpart/schema.sql	Test database schema
lsstpart/stripe_vars.bash	Variables used by stripe testing scripts
lsstpart/test_chunks.bash	Test script for the coarse chunking approach
lsstpart/test_fine_chunks.bash	Test script for the fine chunking approach
lsstpart/test_funs.bash	Common test functions
lsstpart/test_regions.bash	Ra/dec boundaries for the test FOVs
lsstpart/test_stripes.bash	Test script for the stripe approach with <i>ra</i> indexing

<code>lsstpart/test_zones.bash</code>	Test script for the stripe approach with ( <i>zoneId</i> , <i>ra</i> ) indexing
---------------------------------------	---

## Partitioning Approaches

### Stripes

In this approach, each table partition contains *Objects* having positions falling into a certain declination range. Since a FOV will usually only overlap a small ra range within a particular stripe, indexes are necessary to avoid a table scan of the stripe when reading data into memory.

There are two indexing strategies under consideration here - the first indexes each stripe on ra (and also clusters data on ra). The second indexes (and clusters) each stripe on (zoneId, ra). In both cases the height of a stripe is 1.75 degrees. The height of a zone is set to 1 arcminute.

### Chunks

In this approach, a table partition corresponds to an ra/dec box - a chunk - on the sky. See `chunkgen.cpp` for details on how chunks are generated. Basically, the sky is subdivided into stripes of constant height (in declination), and each stripe is further split into chunks of constant width (in right ascension). The number of chunks per stripe is chosen such that the minimum distance between two points in non adjacent chunks of the same stripe never goes below some limit.

There are no indexes kept for the on-disk tables at all - not even a primary key - so each chunk table is scanned completely when read. Two chunk granularities are tested: the first partitions 1.75 degree stripes into chunks at least 1.75 degrees wide (meaning about 9 chunks must be read in per FOV), the second partitions 0.35 degree stripes into chunks at least 0.35 degrees wide (so about 120 chunks must be examined per FOV).

So with coarse chunks, each logical table (Object, DIASource) is split into 13218 physical tables. With fine chunks, 335482 physical tables are needed. Due to OS filesystem limitations, chunk tables will have to be distributed among multiple databases (this isn't currently implemented).

## Testing

### Hardware

- SunFire? V240
- 2 UltraSPARC IIIi CPUs, 1503 MHz
- 16 GB RAM
- 2 Sun StoreEdge? T3 arrays, 470GB each, configured in RAID 5, sustained sequential write speed (256KB blocks) 150 MB/sec, read: 146 MB/sec
- OS: Sun Solaris sun4x\_510
- MySQL: version 5.0.37

## General Notes

Each test is run with both "skinny" *objects* (USNO-B with some additions, ~100bytes), and "fat" *objects* (USNO-B plus 200 DOUBLE columns set to random values) that match the expected row size (including overhead) of the LSST Object table. Any particular test is always run twice in a row: this should shed some light on how OS caching of files affects the results. Between sets of tests that touch the same tables, the `bustcache` program is run in an attempt to flush the operating system caches (it does this by performing random 1MB reads from the USNO-B data until 16GB of data have been read).

## Test descriptions

- Read Object data from the test FOVs into in-memory table(s) with no indexes whatsoever.
- Read Object data from the test FOVs into in-memory table(s) that have the required indexes created before data is loaded. These indexes are:
  - ◆ a primary key on *id* (hash index) and
  - ◆ a B-tree index on *zoneId* for fine chunks, or (for all others) a composite index on (*zoneId*, *ra*).
- Read Object data from the test FOVs into in-memory table(s) with no indexes whatsoever, then create indexes (the same ones as above) after loading finishes.

Note that all tests except the fine chunking tests place *objects* into a single `InMemoryObject` table (and *DIA sources* into a single `InMemoryDIASource` table) which are then used to for cross-matching tests. The fine chunking tests place each on-disk table into a separate in-memory table. This complicates the crossmatch implementation somewhat, but allows for reading many chunk tables in parallel without contention on inserts to a single in-memory table. It allows crossmatch to be parallelized by having different clients call the matching routine for different sub-regions of the FOV (each client is handled by a single thread on the MySQL server).

The following variations on the basic crossmatch were run (all on in-memory tables):

- Test performance using both match-orders: *objects* to *DIA sources* and *DIA sources* to *objects*
- Test both slim and wide matching:
  - ◆ a slim match stores results simply as pairs of keys by which *objects* and *DIA sources* can be looked up
  - ◆ a wide match stores results as a key for a *DIA source* along with the values of all columns for the matching *object*. This is important for the fine chunk case, since looking up objects becomes painful when they can be in one of 100+ in-memory tables.

## Performance Results