# sqlLoader

# A Distributed Workflow System for Data Import, Validation, Publication and Curation

Ani R. Thakar (JHU,
Alex Szalay (JHU),
Jim Gray (Microsoft Research)
Maria Nieto-Santisteban (JHU)

May 8, 2004

## 1 Introduction

The Sloan Digital Sky Survey (SDSS) is a multi-institution, internationally funded project to map about half of the northern sky in unprecedented detail with a dedicated 2.5m telescope and special purpose instruments [SDSS]. The project began taking observations in 2000 and is scheduled for completion in 2006. The resulting digital archive – the SDSS Science Archive – will contain the distillation of the calibrated scientific data from the survey. The raw image data is expected to be about 40TB in size and the catalog data (Science Archive) is expected to be about 2-3 TB in size. Both will be publicly distributed for online access.

The SDSS has had three data releases thus far: the Early Data Release (EDR) in June 2001, Data Release 1 (DR1) in June 2003, and Data Release 2 (DR2) in March 2004. DR2 represents a public distribution of about 2 TB of catalog data in the form of online databases served by the SDSS Catalog Archive Server (CAS). The online web interface that provides access to the CAS is called SkyServer (http://skyserver.sdss.org/). A commercial relational database management system (RDBMS) – Microsoft's SQL Server – is used to store and serve up the data to the scientific community. The DBMS provides advanced data searching and query optimization capabilities, but we also built a multi-dimensional spatial indexing scheme – the Hierarchical Triangular Mesh (HTM) – into the DBMS to enable fast spatial searches and data mining. The DR2 SkyServer site also provides an ability to submit batch queries and to build private databases of query results [O'Mullane03, Nieto03] .

Loading such large multi-dimensional datasets into the DBMS is a critical and time-consuming step that is fraught with potential missteps. In many ways, loading the data is the weakest link in archive publishing. It is typically the *most error-prone and time-consuming* yet *least planned and budgeted* part of archive operations.

.

The Virtual Observatory (VO) [Szalay01] also demands integrating many historical and current datasets into online repositories. Loading these datasets (and reloading them as schemas change) will be an ongoing task. With the multi-terabyte to petabyte data volumes anticipated for the upcoming archives in the VO, time is also a critical factor. Disk speeds will not keep up with the increase in data volume, and in general the loading of such large archives will take several days if parallelism is employed, several weeks if not.

In practice, archive data often has to be reloaded several times even before it is published: errors in the data cause reloads, errors in the data-processing pipeline cause reloads, and sometimes changes to the schema or performance considerations make reloading the data to be the most expedient and clean option. The efficiency and reliability of the data loading process therefore is of paramount importance in order to:

- minimize the time and supervision required to reload the data, and
- get the data correct the first time it is published.

A crucial fact about public archives is that once the data is published, it is immutable, i.e., it cannot be retracted or changed, especially after science has been based on the data and papers have been published. As such, we only get one chance to get the data right, so the loading procedure must be as *reliable*, *thorough*, and as *automated* as possible. Accordingly, we have invested a great deal of time and effort in developing the data loading pipeline for the SDSS data.

We describe this pipeline here, and discuss how it could be adapted as a general-purpose loading tool for other multi-terabyte and petabyte digital archives that will be coming online in the near future.

## 2 FITS-to-CSV Converter

The raw data stored in the SDSS OpDB (operations database) is exported in the form of FITS files. There are several different types of FITS files exported, corresponding to distinct datasets within the SDSS data. There is both image and spectroscopic data. A unit of image data exported is termed a *chunk*, and corresponds to a single resolve operation in the OpDB. As the data is often recalibrated as the image processing pipeline is refined, different calibrations result in different *skyVersions* of the image data:

- **Target** skyVersion – the calibration from which the spectral targets were chosen.
- **Best** skyVersion – the latest, greatest calibration of the data.
- **Runs** skyVersion – a temporary calibration that may be reclassified as target or best later, but until then is only for internal collaboration use. It is not released to the public.

In addition to the image datasets, there are spectroscopic plates and tiling data that also are included in the public data release. These FITS files are converted into ASCII CSV (comma-separated values) files and JPEG image files for ingestion into the SQL databases, by a utility called *sqlFits2Csv* that knows about the database schema that the files are destined for, so it creates one type of output file for each table in the database. This utility also assigns to each catalog object a unique 64-bit object-ID that is used as the primary key in the corresponding database table.

The conversion to CSV also acts as a blood-brain barrier between the Linux and Windows world and between the file and database world. All the popular database systems support a generic CSV file converter. The CSV files are SAMBA-mounted on the windows side and imported into the databases by the sqlLoader distributed workflow system described below. The data flow is shown in Fig. 1.

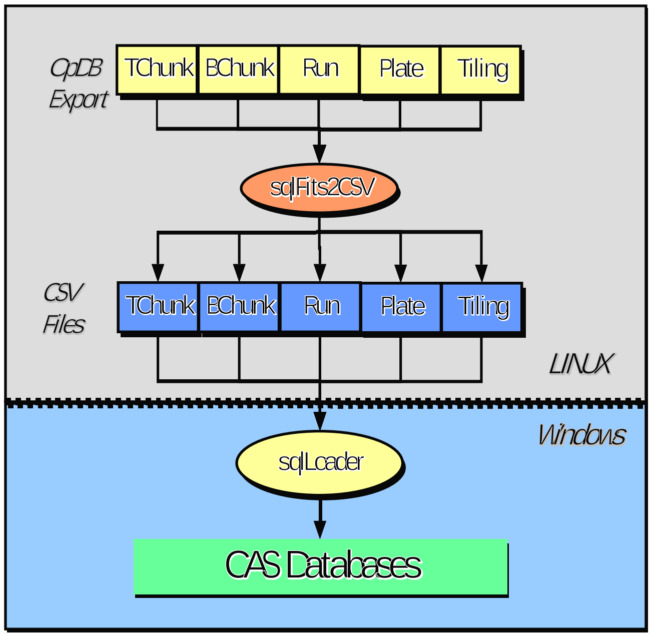


**Figure 1. SDSS Data Import pipeline.** FITS data exported by the OpDB is first converted to CSV format on the Linux side. It is then Samba-mounted and loaded into the SQL Server databases on the Windows side.

## 3 Workflow Management

The loader workflow is controlled via the SQL Server Agent process. There are two agent jobs created in order to control the workflow – the **Load** job and the **Pub** job. The load job runs once every minute and picks up the next step from the Load stage (see below) for the first active task and executes it. Similarly, the Pub job wakes up once every minute and picks up the next step in the Publish-Merge or Finish stages. The workflow stages are shown in Figure 2.

There are three distinct stages in the loader workflow:

1. LOAD – this stage includes checking the input CSV files, loading them into temporary task DBs, validating the data in the task DBs and making a backup of the task DB.
2. PUBLISH – Publishing the individual task DBs to the publish DB; merging of the various data streams (imaging, spectro, tiling) in the publish DB and creating the indices on the main tables.
3. FINISH – Running the final tests, creating the derived tables and pre-computed joins, and the corresponding indices.

Stages 2 and 3 must be run in sequential at the moment, since they all write to the same (publish) DB. The Load-Validate can be run in parallel on a cluster of load servers, each having its own view of the schema and its own task DB.

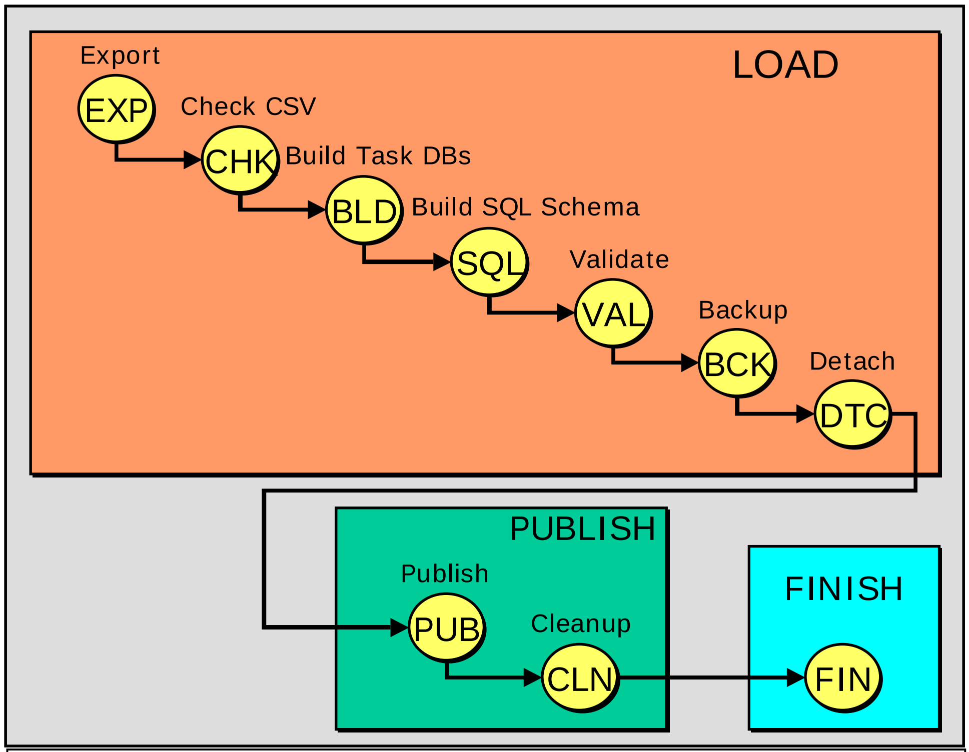


**Figure 2. The workflow diagrams for the sqlLoader,** showing the LOAD, PUBLISH and FINISH workflows. The LOAD workflow is the most time-consuming and it can be run in distributed parallel mode. The PUBLISH and FINISH must be done sequentially.

## 4 Distributed Framework

The framework for the sqlLoader pipeline is divided into two parts: **loadadmin** and **loadsupport**. This facilitates distributed and parallel data loading with a cluster of loadservers. The *loadadmin* scripts and databases control the load framework on the *master* loadserver. The *loadsupport* scripts and DBs set up the loading on the *slave* loadservers and build the ancillary framework to facilitate the loading process, such as setting up the user privileges, setting up links between master and slave loadservers (if applicable), setting up the spatial indexing, and utilities to manage units of the loading process (phase, step, task etc.). Figure 3 shows the relationship between the loadadmin and loadsupport parts of the framework. Each of the satellite or slave servers links only to the master, not to each other.

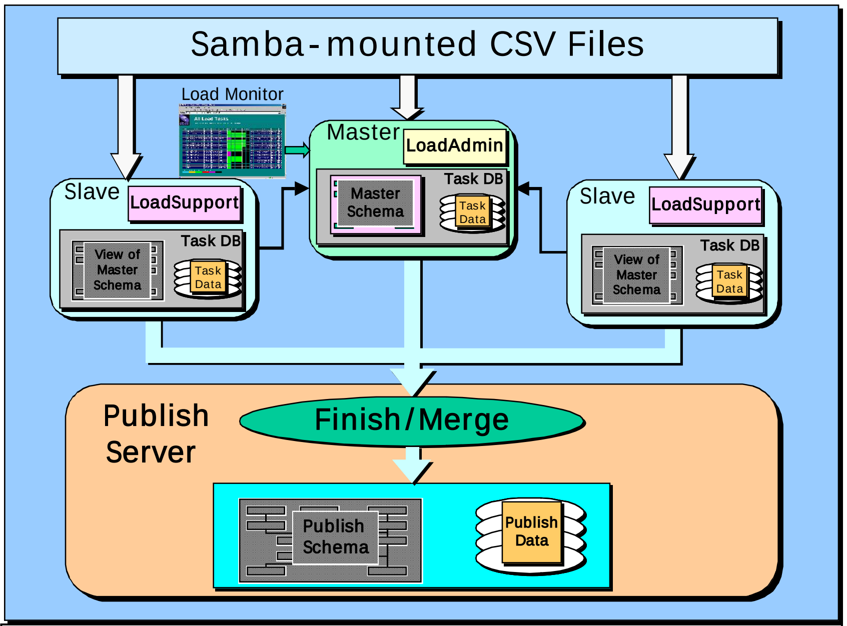


**Figure 3. Schematic of relationship between the master (loadadmin) server and the slave (loadsupport) servers.**

The loadsupport database contains read-only remote *views* of the loadadmin database rather than copies, so that changes in the master data or schema are automatically reflected on the slaves. This uses SQL Server'slinked server mechanism to make remote databases and tables appear to be local.

### 4.1 *Load* and *Publish* Roles

The loadadmin and loadsupport servers have different roles in the distributed implementation: the loadadmin server runs both in *load* and *publish* roles, whereas the loadsupport servers run only in the *load* role. The load role corresponds to the loading of the CSV files into the individual component databases (one per load unit), whereas the publish role involves the validation and transfer of the component database contents into the publish DB for each dataset. The loading is therefore done in parallel, and the publishing brings all the loaded components together into the publish DB on the master server. We refer to this final publish step as the merge/finish step to avoid confusion with the copying of the task DBs to the publish DB (see below).

The **sqlLoader** framework contains several kinds of scripts:

1. Windows command scripts (*.bat* files) that run the loading framework and SQL scripts.
2. SQL scripts (*.sql* files) that contain the SQL code to set up the loading framework and load the data.
3. DTS (Data Transformation Services) packages to perform special-purpose data import tasks.
4. Visual Basic (VBScript) scripts that are used for syntax-checking the CSV files and parsing the documentation files to load them into the databases.

## 4.2 Loader Command Scripts

The initial creation of the loader framework, including building the loadadmin and loadsupport databases, must be done via command scripts that perform the database creation and other system level setup tasks that need to be performed only once when the pipeline is first configured. There are two main scripts – one to build the loadadmin (master) framework, and the other to build the loadsupport (slave) framework. In the case where a single loadserver is used for the data loading and publishing, both of these would be constructed on the same machine.

1. **build-loadadmin.bat** - This builds the loadadmin framework on the master server. This is to be run only once from the command line on the main (master) loadserver machine. It takes no parameters. It performs the following tasks:
   - gets the loadserver host name
   - creates the path for the loader logs
   - looks for local log file and deletes it
   - checks whether the directory where the loadadmin database is to be created exists, and if it does not exist, creates it.
   - makes load logs world-readable
   - sets up 2 network shares – for the **root** of the sqlLoader directory structure and the **loadlog** directory that holds the logs. This is necessary so that the slave servers can see these directories.
   - runs the following SQL scripts:
     - **loadadmin-build.sql** – creates the loadadmin database.
     - **loadadmin-schema.sql** – installs the schema in the loadadmin database.
     - **loadadmin-local-config.sql** – sets up the local configuration for the master loadserver. This script has to be updated manually with the local information.
2. **build-loadsupport.bat** - This script must be run on each of the loadservers in the configuration, including the master. It builds the loadsupport part of the framework on the current loadserver (which can be the master or one of the slaves

if applicable), including the **loadsupport database**, the HTM, loadsupport utilities etc. This script performs the following functions:

- sets the name of the master loadserver
- sets up the logging paths for this loadserver
- sets up the load-support environment on this server
- builds the load-support DB and schema on this server
- sets up the load-support stored procedures and utlilities
- sets up this loadserver's <u>role</u> - ***loader*** or ***publisher*** or both: a loader only stuffs the data in the databases, but does not run the *validation* and *publish* steps
- sets up the HTM

## 4.3 Loader SQL Scripts

The command scripts invoke SQL scripts to carry out the actual database tasks. These are the loadadmin and loadsupport SQL scripts described in the table below. Most of the functions required for the data import, validation, and publishing are coded in the form of SQL stored procedures in the loadsupport, task and publish databases. These are defined in the following scripts and installed in the appropriate databases upon creation.

| *Name of script, remarks* | *Specific functions performed by script* |
|---|---|
| **loadadmin-build.sql** | • Deletes the existing **loadadmin** DB if any.<br>• Turns on the trace flag 1807 - <u>this is a secret Windows flag that allows us to mount remote DBs.</u><br>• Sets a bunch of DB options.<br>• Turns **autoshrink** off - this is very important otherwise the performance bogs down when autoshrink tasks run in the background. |
| **loadadmin-schema.sql** | • Creates Task and Step tables in the loadadmin DB.<br>• Inserts NULL task and step - this is necessary so we can assign system errors if everything fails.<br>• Creates NextStep table - this drives the sequence of loading by specifying what are the procedures for the next step.<br>• Creates ServerState table - this allows us to stop the server so that processing is stopped.<br>• Creates Constants table and put it in all the paths<br>• Gets server name from global variable (e.g. sdssad2) - Note: SQL name for the server must be the same as the Windows name. |
| **loadadmin-local-config.sql** <u>Must be adapted for the local configuration</u> before running | • Sets up the paths for the CSV files.<br>• Sets up backup paths.<br>• Sets up the **loadagent** user and domain so that the |

| | |
|---|---|
| the loader. | SQL Agent can be started up. |
| **loadsupport-build.sql** Needs to be edited to update the domain account names explicitly. In the future, will likely be reorganized to deal differently with master/slave loadservers. | • Creates a "webagent" user account which is used by the load monitor web interface to connect to the loadserver(s) and run loader tasks<br>  o make sure that webagent is a sysadmin on the loadserver<br>  o make sure only the master loadserver does this. |
| **loadsupport-schema.sql** | Creates all the tables in the schema. |
| **loadsupport-loadserver.sql** Need this to allow SQL scripted access. | Creates a single table with a single row which is the name of this loadserver. |
| **loadsupport-link.sql** Sets up the link between the master and slave server for this slave. | • Creates a 2-way link-server relationship between this slave server and the master server.<br>• Sets up all the views.<br>• Enables remote transactions. |
| **loadsupport-sp.sql** Sets up the stored procedures for loading from this server. | • Constructors for **phase**, **step**, and **task.**<br>• Start/end steps - this is done only on the loadserver.<br>• Kill task, ensuring that:<br>  o log records are kept<br>  o files are cleaned up.<br>• DB is deleted only when the same task is re-submitted (with new taskID). |
| **loadsupport-utils.sql** | Pre-load utilities |
| **loadsupport-steps.sql** Controls the high-level steps | Each step has the following:<br>• A stored procedure with the name "sp<*name-of-step*>step" associated with it - which is the meat of the step's logic.<br>• A "sp<*name-of-step*>" procedure that is a wrapper that calls the sp<*name-of-step*>step procedure. |
| **loadsupport-show.sql** | • Displays load monitor screens |

## 4.4 Schema Files

The schema creation scripts for the CAS databases are in the **schema** subdirectory. The contents of this subdirectory encapsulate the data model for the archive, and hence will change the most when it is adapted to other (non-SDSS) archives. There are 5 subdirectories at this level:

1. **csv** - contains CSV (comma-separated values) outputs from the documentation generation scripts;  in general this directory contains input files for metadata tables.
2. **doc** - the documentation content files are here.
3. **etc** - miscellaneous SQL script files are here, for housekeeping and utility schema-related functions.
4. **log** - the weblogging scripts are here.
5. **sql** - this is the main subdirectory containing the schema files.  The various schema tables, views, stored procedures and functions are created by the SQL scripts in this subdirectory.  The following files are here:
    - boundary.sql - Creates the tables and funtions related to boundaries and polygons
    - constantSupport.sql - Creates the support functions for various constants and enumerated types
    - dataConstants.sql - Sets the values of the constants and enumerated types
    - metadataTables.sql - Creates the tables that describe the data tables
    - myTimeX.sql - Contains scripts for performance measurements
    - nearFunctions.sql - Creates the various functions that find nearby objects
    - photoTables.sql - Creates the **imaging** (photo) tables
    - spBackup.sql - Creates stored procedures to back up databases
    - spectroTables.sql - Creates the **spectro** and **tiling** tables
    - spFinish.sql - Contains the stored procedures for the **Finish** step in the loading/publishing
    - spGrantAccess.sql - Creates the stored procedure that sets the privileges correctly for users after databases are loaded
    - spHTMmaster.sql - Creates the stored procedures to install HTM into the **master** database
    - spHTM.sql - Creates the stored procedures to install HTM into other DBs
    - spManageIndices.sql - Creates the stored procedures for managing the index creation
    - spPublish.sql - Contains the stored procedures for the **Publish** step in the loading/publishing
    - spSetValues.sql - Contains the stored procedure that sets and updates column values after the bulk loading
    - spValidate.sql - Contains the stored procedures for the **Validate** step in the loading/publishing
    - views.sql - Defines and creates the various views on the data tables
    - webSupport.sql - Creates the stored procedures to support the web (HTTP) interfaces to the DBs, including those needed to execute SQL queries submitted via the skyServer and the sdssQA
    - zoomTables.sql - Creates the schema for the Zoom-related tables

## 5 The Data Validator

Data emerges from the pipeline as ASCII CSV (comma separated values) files and image files in JPEG and GIF format.   Each batch of files is imported into a staging database where it is validated and enhanced. Once validated the data is moved to the production or archive databases. The validation steps are shown in Figure 4.

Pipeline → SQL validation in Staging DB → {Best, Target, Run} SQL databases

Photometric and spectroscopic data have different routines, but the Photo Best, Target, and Runs datasets have approximately the same validation logic.

## 5.1 spValidator

The validator is invoked as a stored procedure on a particular database. Its job is to validate that database. It searches the Task table using the host name and DB-name as a key. The returned record tells the validator:

- The type of validation (photo or spectro).
- If it is a photo job, the subtype (BEST, TARGET, RUNS).
- The destination database.
- A job ID that is used to key all future log events.

The spValidator then branches to spValidateSpectro or spValidatePhoto. When these routines complete, spValidator writes a completion message in LoadSupport.dbo.Task and exits. At each step the validation routines record the result of a test. The loader interface can watch the validation progress and can assess the success or failure of the validation by looking at this journal (in the load database) and by looking at the job summary record.

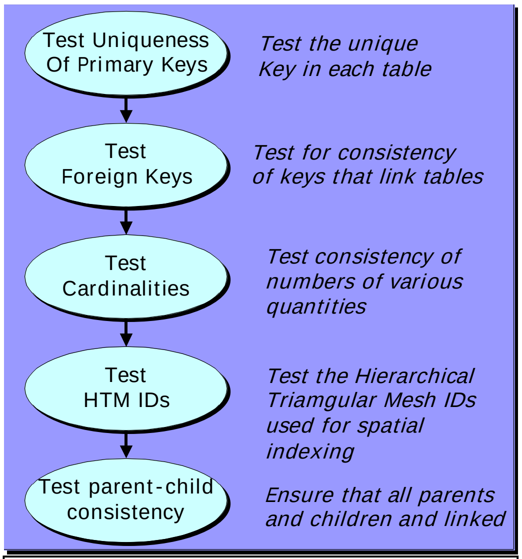

Test Uniqueness Of Primary Keys
Test the unique Key in each table
Test Foreign Keys
Test for consistency of keys that link tables
Test Cardinalities
Test consistency of numbers of various quantities
Test HTM IDs
Test the Hierarchical Triamgular Mesh IDs used for spatial indexing
Test parent-child consistency
Ensure that all parents and children and linked

**Figure 4. Data validation checks performed by the Validator**

## 5.2 spValidatePhoto

The photo validator performs the following checks:

1. It checks the uniqueness of the following primary key fields:

```
Chunk.chunkNumber
CrossID.(objID,surveyID,surveyObjID)
First.objID
Field.FieldID
FieldProfile.(fieldID, bin, band)
Frame.FieldID
PhotoObj.objID
PhotoProfile(objID, bin, band)
PhotoZ.objID
Rosat.objID
Segment.segmentID
StripeDefs.stripe
```

```
Survey.surveryID
Synonym.(objID, matchID)
USNO.objID
```

2. It creates the following temporary indices to make subsequent tests run faster:
```
    PhotoObj(objID),
    Field(FieldID),
    PhotoObj(HTMid, ObjID,cx,cy,cz,type, status)
```

3. It then tests the following foreign keys:

| | | |
|---|---|---|
| `Chunk.stripe` | ⇒ | `StripeDefs.stripe` |
| `CrossID.objID` | ⇒ | `PhotoObj.objID` |
| `CrossID.surveyID` | ⇒ | `Survey.SurveyID` |
| `Field.segmentID` | ⇒ | `Segment.segmentID` |
| `FieldProfile.fieldID` | ⇒ | `Field.fieldID` |
| `First.objID` | ⇒ | `PhotoObj.objID` |
| `Frame.fieldID` | ⇒ | `Field.fieldID` |
| `PhotoObj.fieldID` | ⇒ | `Field.fieldID` |
| `PhotoProfile.objID` | ⇒ | `PhotoObj.objID` |
| `PhotoZ.objID` | ⇒ | `PhotoObj.objID` |
| `Rosat.objID` | ⇒ | `PhotoObj.objID` |
| `Segment.chunkNumber` | ⇒ | `Chunk.chunkNumber` |
| `Synonym.ObjID` | ⇒ | `PhotoObj.objID` |
| `Segment.stripe` | ⇒ | `StripeDefs.stripe` |
| `USNO.objID` | ⇒ | `PhotoObj.objID` |

4. Next, it checks to see if the advertised populations match the real populations:
```
      Segment.nFields   = count(fields) group by segmentID
Field.nObjects    = count(PhotoObj) group by fieldID
PhotoObj.Nprofiles = count(PhotoProfile) group by objID
```

5. It then looks at the PhotoObj parents (who was deblended from whom) and tests to see that `PhotoObj(nChild) = count(PhotoObj)` with that Parent for  for the first 1000 non-null parents.

6. It also tests the first 1,000 to see that the external HTM calculation is similar to the internal one (a few errors are allowed due to rounding, but 99% of the results should agree exactly).
```
    Frame.htmID
    Mosaic.htmID
    PhotoObj.htmID
```

7. Next it computes the neighbors of each object.  For a Best database a 30 arc second neighborhood is computed.  Target and Runs databases use a 3 arcsecond radius.  The neighbors computation is complex enough that it has its own writeup (spNeighbors) as a separate memo.  But the idea is the following:
    a. the zone table is built from PhotoObj
    b. it is augmented with “visitors” from the target area who might contribute neighbors.
    c. the margins are added in

   d. the 3 zone-joins are done to compute the neighbors.
   e. the zone and foreigners tables are dropped.

This is the longest step of the validation process.

8. Lastly, the validator drops the indices it created for the validation work, namely `PhotoObj.I, PhotoObj.HTM, Field.I`

spValidatePhoto then returns to spValidate. If there is spectroscopic data in the database, it will be validated next.

### 5.3 spValidateSpectro

Testing spectroscopic data is simpler. This data is always destined for the Spectro part`of the database schema and there are many fewer tests.

1. spValidateSpectro first tests the uniqueness of the primary keys.
   ```
   Plate.plateID,
   SpecObjAll.SpecObjID
   ELRedshift.ELRedshiftID
   XCRedshift.XCRedshiftID
   SpecLine.SpecLineID
   SpecLineIndex.SpecLineIndexID
   ```

2. It then creates two indices to make the subsequent tests run much faster: `SpecObjAll(SpecObjID), Plate(PlateID)`

3. It then tests the following foreign keys:
   ```
   SpecObjAll.plateID          ⇒    Plate.plateID
   ElRedshift.specObjID        ⇒    SpecObjAll.specObjID
   SpecLine.specObjID          ⇒    SpecObjAll.specObjID
   SpecLineIndex. specObjID    ⇒    SpecObjAll.specObjID,
   XCredshift.specObjID        ⇒    SpecObjAll,specObjID.
   ```

4. It also tests the first 1,000 HTM IDs in the SpecObj table (`SpecObj.htmID`) to check that the external HTM calculation is similar to the internal one.

5. Lastly it drops the working indices and returns.

## 6 Setting Up the Loader Framework

The following steps must be performed to set up and install the sqlLoader so that data can be loaded into the databases. Most of these steps (#1-9) are things that typically only have to be executed once, the first time that the loader is installed on the loadserver. If the loadserver has already been configured, proceed directly to step 10.

1. Set the SQL Server security on the master loadserver (loadadmin) machine to **mixed security**, i.e. Windows *and* SQLServer authentication. This is necessary for the *webagent* user to be able to connect to the loadadmin server.

2. Ensure that the SQL Server Agent is running on the master loadserver (*loadadmin*).
3. On the *loadadmin* machine, check out a copy of the **sqlLoader** module in the C:\ drive.
4. Make sure that the **tempDB** properties are set up correctly. In Enterprise Manager under the Databases tab, select tempDB and then select Taskpad from the View menu. This will show the DB properties at a glance. You need to check/set the following:
    - The data file for the DB on the D: drive needs to be set to grow automatically.
    - Make sure that the data file for the DB on the C: drive is not set to grow automatically.
    - The size of the data file should be at least 10GB (10000MB), preferably 20GB (20000MB) if there is room to spare. You can only increase the size of the data files from the initial size (you cannot decrease the size below the initial size the DB was created with).

    Once you are done, press the Ok button at the bottom to apply the changes. It will take a while to increase the size of the data file (few minutes to half hour).

5. Set up the load monitor web interface on a Windows webserver (running IIS):
    1. Copy/move the **admin** subdirectory of sqlLoader to the web tree where the admin pages will be served from.
    2. Set up a virtual directory in IIS to point to this directory.
    3. Modify the **connection.js** file (edit with Notepad) in the admin directory to replace the xxx-ed out password for the **webagent** user with the real password, and change the Data Source to point to the master loadserver machine.
6. Tweak the **loadadmin/loadadmin-local-config.sql** file as per the local configuration parameters (see Loader Admin Framework section above). Make sure that all the directory paths that are specified in this file actually exist! For the backup directory path, also make sure that the Sharing (not NTFS Security) privileges for the backup directory allow Full Control for Everyone.
7. Create a share for the master sqlLoader directory on the web server and the slave servers.
8. Edit **loadadmin/set-loadserver.bat** to set the name of the master loadserver machine.
9. Edit the **loadadmin/loadsupport-build.sql** file to update the domain account names, if necessary.
10. If this is not the first time, and a previous loadadmin/loadsupport environment exists on this machine, delete it by doing the following:
    - Kill any tasks that may be running in the previous Load Monitor.
    - In Enterprise Manager, go to the local SQL Server Group and open the Databases tab. Then delete each of he following databases by right-clicking the mouse on the database and selecting Delete to delete it:

    - The publish DBs, called <*export type*><*dataset*>, e.g., BESTTEST and TARGTEST or BESTDR1 and TARGDR1
    - Any temporary load DBs <*dataset*>_<*export type*><*xid*>, e.g. TEST_BEST1_35_471938
    - The **loadadmin** DB.
    - The **loadsupport** DB.

11. Run the following scripts from a command shell (Start->Run...->cmd). Note that step 4 should be run on each loadserver in the configuration (including the master), each of which will have the sqlLoader directory shared.

    - **C:**
      Make sure you are on the C: drive.
    - **cd C:\sqlLoader\loadadmin**
      Go to the loadadmin subdirectory in the loader.
    - **build-loadadmin.bat**
      This will create the loadadmin environment and DB.
    - **build-loadsupport.bat -LP** (on loadadmin/master)
      **build-loadsupport.bat -L** (on each loadserver/slave other than master)
      Run this on each loadserver in the configuration. On all but the master (loadadmin) server, the sqlLoader directory is shared. This will create the loadsupport environment and DB and set this loadserver's role to both LOAD and PUBLISH. If the last message from running the above 2 steps says "Access denied..." rather than "1 file(s) moved", this means that the log file could not be moved to the loadlog directory. You will need to right-click on the C:\loadlog directory, select Properties or Sharing and Security and go to the Sharing tab. In the Sharing permissions screen, select Everyone and turn on Full Control. After doing this, you will have to restart the setup at step 9 above.
    - **build-publish-db.bat <*dataset*> <*db-data-size*> <*db-logsize*>**
      This script should be run on the master only. It will automatically create publish DBs for the BEST and TARGET skyversions for each dataset. This will take some time to run for multi-GB sizes. The sizes you pick for the DB and log files should be approximately:

      **<*db-data-size*>** = the total size of the input data + 50%, in MB.
      **<*db-log-size*>** = at least 50% of the data size, in MB. e.g.,

      **build-publish-db.bat TEST 20000 10000**

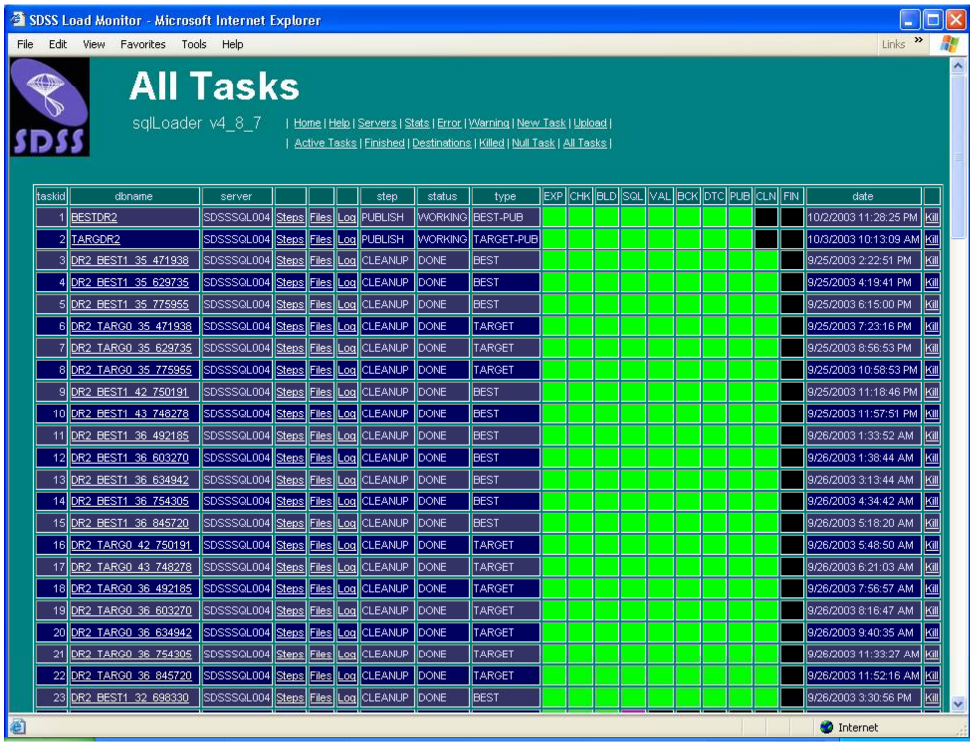
      These are just the initial sizes - SQL Server will expand the file sizes as needed, but this comes at a performance cost so it's better to allocate enough space to begin with.

## 7 Load Monitor or Admin Webpage (admin/ subdir)

The **admin** subdirectory contains the web pages and associated files and scripts for the loader administration web interface (the **Load Monitor**). This subdirectory should be copied or moved to the web tree where the load monitor is to be accessed from. Note that currently the admin web pages must be installed on a Windows machine (running IIS) since they use ASP (active server pages) technology. The security setup is Windows authentication at the moment and this requires that the pages be accessed also from a Windows machine. The web-server connects to the loadsupport DB of the loadserver.

There are 4 kinds of files in this directory:

1. Active Server Pages (*.asp* files) that correspond to actual web pages.
2. Cascading Style Sheets (*.css* files) that are used by the web pages to set up the look and feel.
3. Javascript (*.js*) files that contain functions in jscript to perform loader admin procedures.


SDSS Load Monitor - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
SDSS
All Tasks
sqlLoader v4_8_7
| Home | Help | Servers | Stats | Error | Warning | New Task | Upload |
| Active Tasks | Finished | Destinations | Killed | Null Task | All Tasks |
Internet

| taskid | dbname | server | | | | step | status | type | EXP | CHK | BLD | SQL | VAL | BCK | DTC | PUB | CLN | FIN | date | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | BESTDR2 | SDSSSQL004 | Steps | Files | Log | PUBLISH | WORKING | BEST-PUB | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | ■ | 10/2/2003 11:28:25 PM | Kill |
| 2 | TARGDR2 | SDSSSQL004 | Steps | Files | Log | PUBLISH | WORKING | TARGET-PUB | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | ■ | 10/3/2003 10:13:09 AM | Kill |
| 3 | DR2_BEST1_35_471938 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 2:22:51 PM | Kill |
| 4 | DR2_BEST1_35_629735 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 4:19:41 PM | Kill |
| 5 | DR2_BEST1_35_775955 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 6:15:00 PM | Kill |
| 6 | DR2_TARG0_35_471938 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 7:23:16 PM | Kill |
| 7 | DR2_TARG0_35_629735 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 8:56:53 PM | Kill |
| 8 | DR2_TARG0_35_775955 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 10:58:53 PM | Kill |
| 9 | DR2_BEST1_42_750191 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 11:18:46 PM | Kill |
| 10 | DR2_BEST1_43_748278 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/25/2003 11:57:51 PM | Kill |
| 11 | DR2_BEST1_36_492185 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 1:33:52 AM | Kill |
| 12 | DR2_BEST1_36_603270 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 1:38:44 AM | Kill |
| 13 | DR2_BEST1_36_634942 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 3:13:44 AM | Kill |
| 14 | DR2_BEST1_36_754305 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 4:34:42 AM | Kill |
| 15 | DR2_BEST1_36_845720 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 5:18:20 AM | Kill |
| 16 | DR2_TARG0_42_750191 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 5:48:50 AM | Kill |
| 17 | DR2_TARG0_43_748278 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 6:21:03 AM | Kill |
| 18 | DR2_TARG0_36_492185 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 7:56:57 AM | Kill |
| 19 | DR2_TARG0_36_603270 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 8:16:47 AM | Kill |
| 20 | DR2_TARG0_36_634942 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 9:40:35 AM | Kill |
| 21 | DR2_TARG0_36_754305 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 11:33:27 AM | Kill |
| 22 | DR2_TARG0_36_845720 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 11:52:16 AM | Kill |
| 23 | DR2_BEST1_32_698330 | SDSSSQL004 | Steps | Files | Log | CLEANUP | DONE | BEST | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | ■ | 9/26/2003 3:30:56 PM | Kill |

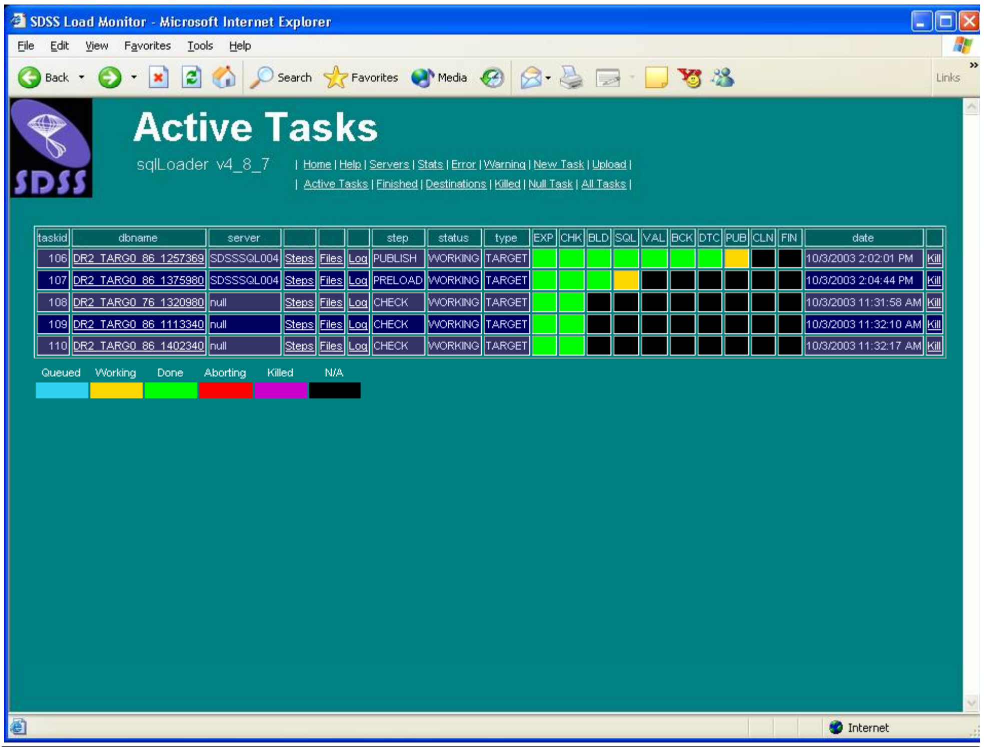
**Figure 5. Load M onitor showing the All Tasks display page.**

4. Include (*.inc*) files that are included in other files above.

There is a **docs.asp** file that contains an overview of the loading process and framework. It is the page linked to the "Help" link on the admin page. The **img** subdirectory contains the JPG and GIF images used in the web pages. There are ASP files corresponding to each of the commands listed in the menu on the main sqlLoader page. Each of these ASP scripts formulates and submits a SQL query to the loadsupport database on the loadadmin server, which sees the query as being submitted by the **webagent** user. The query may execute a stored procedure in the loadsupport DB or request rows from a loadsupport table.

The functions of the ASP scripts are as follows:

1. **default.asp** – this is the main sqlLoader page, showing the current software version number and the menu of available commands.
2. **tasklist.asp** shows the All Tasks display page. Every task that is listed in the Task table regardless of its status is shown in a tabular display (Fig. 3).
3. **activetasks.asp** lists only the currently active tasks from the Task table in tabular format (Fig. 5).
4. **showtask.asp, showsteps.asp, showfiles.asp, showlog.asp** – these scripts display information in different tables corresponding to a particular task: the Task table, the Step table, the Files table and the Phase table (showlog.asp provides a display


SDSS Load Monitor - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Links
Active Tasks
sqlLoader v4_8_7
| Home | Help | Servers | Stats | Error | Warning | New Task | Upload |
| Active Tasks | Finished | Destinations | Killed | Null Task | All Tasks |
Internet

| taskid | dbname | server | | | | step | status | type | EXP | CHK | BLD | SQL | VAL | BCK | DTC | PUB | CLN | FIN | date | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 106 | DR2_TARG0_86_1257369 | SDSSSQL004 | Steps | Files | Log | PUBLISH | WORKING | TARGET | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟨 | ⬛ | ⬛ | 10/3/2003 2:02:01 PM | Kill |
| 107 | DR2_TARG0_86_1375980 | SDSSSQL004 | Steps | Files | Log | PRELOAD | WORKING | TARGET | 🟩 | 🟩 | 🟩 | 🟨 | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | 10/3/2003 2:04:44 PM | Kill |
| 108 | DR2_TARG0_76_1320980 | null | Steps | Files | Log | CHECK | WORKING | TARGET | 🟩 | 🟩 | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | 10/3/2003 11:31:58 AM | Kill |
| 109 | DR2_TARG0_86_1113340 | null | Steps | Files | Log | CHECK | WORKING | TARGET | 🟩 | 🟩 | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | 10/3/2003 11:32:10 AM | Kill |
| 110 | DR2_TARG0_86_1402340 | null | Steps | Files | Log | CHECK | WORKING | TARGET | 🟩 | 🟩 | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | ⬛ | 10/3/2003 11:32:17 AM | Kill |

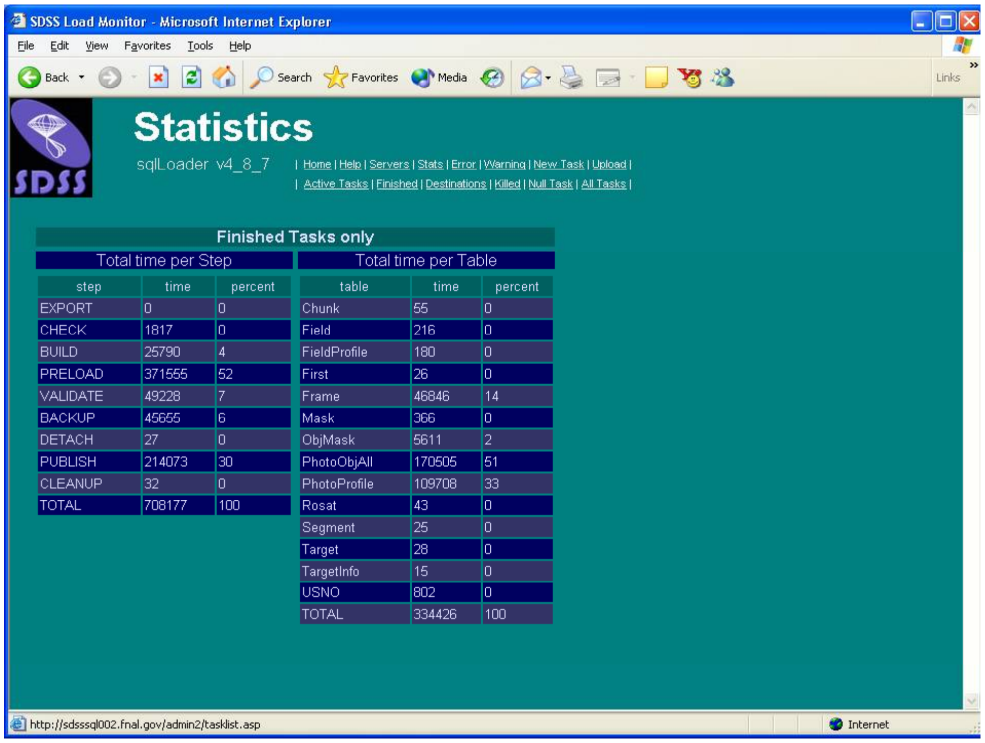
| Queued | Working | Done | Aborting | Killed | N/A |
|---|---|---|---|---|---|
| 🟦 | 🟨 | 🟩 | 🟥 | 🟪 | ⬛ |

**Figure 6. Load Monitor Active Tasks page showing the color-coding used to indicate the status of each step.**

of all the phases for a given task).

In order to set up the admin web pages for the SQL Loader, you need to do the following:

1. Copy/move the **admin** subdirectory of sqlLoader to the web tree on the admin webserver where the admin pages will be served from.


SDSS Load Monitor - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Links
Statistics
sqlLoader v4_8_7
| Home | Help | Servers | Stats | Error | Warning | New Task | Upload |
| Active Tasks | Finished | Destinations | Killed | Null Task | All Tasks |
http://sdsssql002.fnal.gov/admin2/tasklist.asp
Internet

Finished Tasks only

Total time per Step

| step | time | percent |
|---|---|---|
| EXPORT | 0 | 0 |
| CHECK | 1817 | 0 |
| BUILD | 25790 | 4 |
| PRELOAD | 371555 | 52 |
| VALIDATE | 49228 | 7 |
| BACKUP | 45655 | 6 |
| DETACH | 27 | 0 |
| PUBLISH | 214073 | 30 |
| CLEANUP | 32 | 0 |
| TOTAL | 708177 | 100 |

Total time per Table

| table | time | percent |
|---|---|---|
| Chunk | 55 | 0 |
| Field | 216 | 0 |
| FieldProfile | 180 | 0 |
| First | 26 | 0 |
| Frame | 46846 | 14 |
| Mask | 366 | 0 |
| ObjMask | 5611 | 2 |
| PhotoObjAll | 170505 | 51 |
| PhotoProfile | 109708 | 33 |
| Rosat | 43 | 0 |
| Segment | 25 | 0 |
| Target | 28 | 0 |
| TargetInfo | 15 | 0 |
| USNO | 802 | 0 |
| TOTAL | 334426 | 100 |

**Figure 7. Load Monitor Statistics page. Average and cumulative stats for all jobs are displayed.**

2. Set up a virtual directory in IIS to point to this directory.
3. Modify the **connection.js** file (edit with Notepad) to set the password for the **webagent** user, and set the DB server name to the loadadmin server name.

Once the admin web interface is set up in this way, typing the URL corresponding to the admin virtual directory set up in step 2 should bring up the main sqlLoader page that displays the current version number and the menu of available tasks. You are then ready to submit load tasks.

## 7.1 Running the Loader

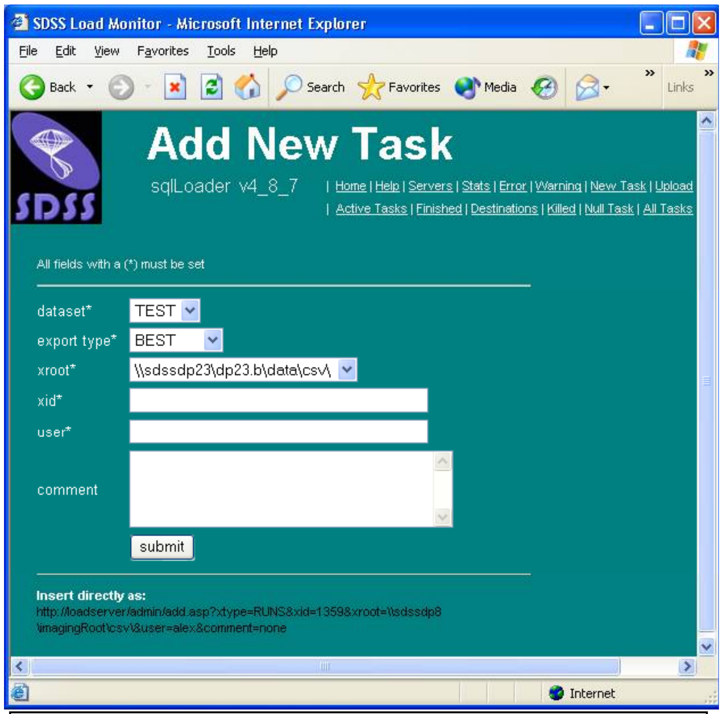
The loading is launched and controlled from the Load Monitor web interface. A new task must be created and launched for each unit of the loading, and this is done using the New Task or the Upload pages.

### 7.1.1 Task Management

The basic unit of loader processing at the top level is a **task**. Tasks are further divided into **steps**, which are in turn subdivided into **phases**. Steps have a well-defined *start* and *end*, whereas a phase does not have a start or end associated with it. A step also has a SQL stored procedure associated with it. The tasks display pages ("Active Tasks", "All Tasks", “Finished” and “Killed”) display task tables containing the taskid, the stepid and the phaseid, hence the granularity of the task display is a single phase.

### 7.1.2 Creating a New Task

The Add New Task page creates a new loading task. The user must enter the following parameters of the task:


SDSS Load Monitor - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Links
SDSS
Add New Task
sqlLoader v4_8_7
| Home | Help | Servers | Stats | Error | Warning | New Task | Upload
| Active Tasks | Finished | Destinations | Killed | Null Task | All Tasks
submit
Internet

All fields with a (*) must be set

dataset* TEST
export type* BEST
xroot* \\sdssdp23\dp23.b\data\csv\
xid*
user*
comment

Insert directly as:
http://loadserver/admin/add.asp?xtype=RUNS&xid=1359&xroot=\\sdssdp8\imagingRoot\csv\&user=alex&comment=none

**Figure 8. Load Monitor Add New Task page.**

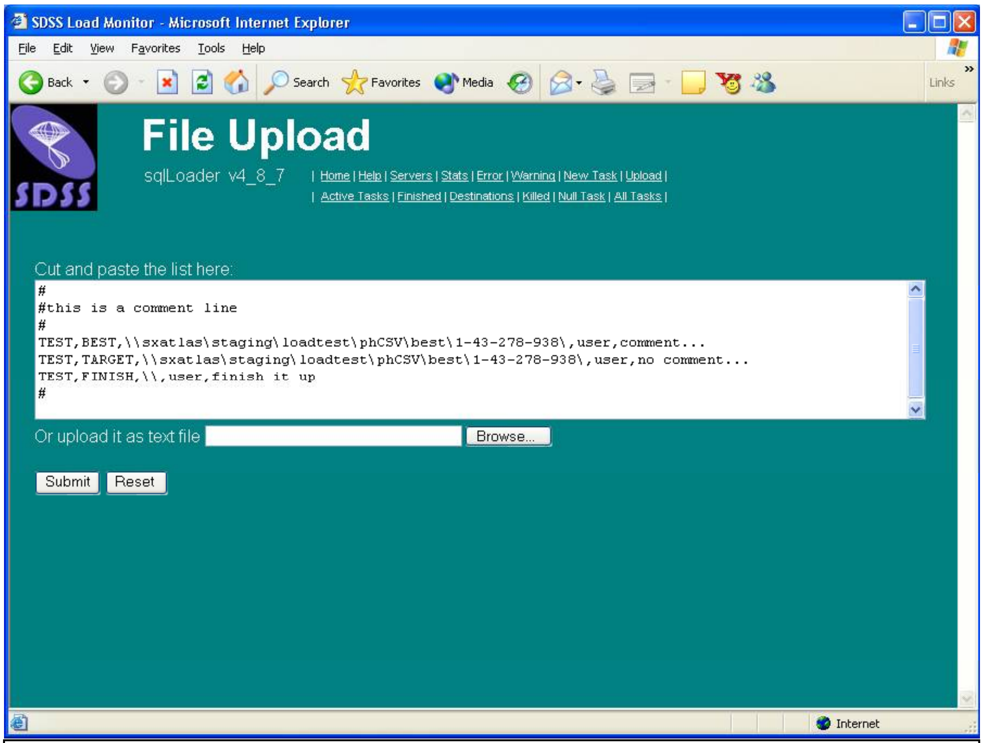
1. **dataset** - this is the release that is being loaded, example DR1, DR2 (or TEST for testing). It is the same as the first parameter given to the **build-publish-db.bat** script.
2. **export type** - the dataset that this is being exported to. The choices are BEST, RUNS of TARGET for an imaging load, PLATES for spectro, TILES for tiling, and a special export type called FINISH for the last step that merges all the data streams (photo, spectro, tiling) and computes the indices.
3. **xroot** - this is the root of the exported CSV directory tree on the LINUX side (Samba-mounted), in Windows notation (\\*hostname**directory**subdir*...).
4. **xid** - the identifier of this export or load unit, i.e. the chunk, plates or tiles that need to be loaded. This is basically the name of the subdirectory in the CSV directory tree that contains the runs, plates or tiles that are to be loaded.
5. **user** - the name of the person who is running this load task.
6. **comment** - an optional comment to describe the purpose or content of this load.

### 7.1.3 Submitting Multiple Load Tasks (File Upload)

Multiple load tasks can be submitted at once by building an upload file containing the task parameters values in CSV format. An example of the contents of a load file are:

```
DR2,BEST,\\sdssdp23\dp23.b\data\csv\phCSV\best\1-82-1176576\,JohnDoe,best34
DR2,BEST,\\sdssdp23\dp23.b\data\csv\phCSV\best\1-86-1184468\,JohnDoe,best35
DR2,BEST,\\sdssdp23\dp23.b\data\csv\phCSV\best\1-86-1257369\,JohnDoe,best36
```

```
    DR2,BEST,\\sdssdp23\dp23.b\data\csv\phCSV\best\1-86-1375980\,JohnDoe,best37
    DR2,BEST,\\sdssdp23\dp23.b\data\csv\phCSV\best\1-86-1402340\,JohnDoe,best38
    DR2,BEST,\\sdssdp23\dp23.b\data\csv\phCSV\best\1-86-1422868\,JohnDoe,best39
    DR2,TARGET,\\sdssdp23\dp23.b\data\csv\phCSV\target\0-32-
676160\,JohnDoe,targ34
    DR2,TARGET,\\sdssdp23\dp23.b\data\csv\phCSV\target\0-32-
462086\,JohnDoe,targ35
    DR2,TARGET,\\sdssdp23\dp23.b\data\csv\phCSV\target\0-34-
705815\,JohnDoe,targ36
    DR2,TARGET,\\sdssdp23\dp23.b\data\csv\phCSV\target\0-37-
477561\,JohnDoe,targ37
    DR2,TARGET,\\sdssdp23\dp23.b\data\csv\phCSV\target\0-37-
581100\,JohnDoe,targ38
    DR2,TARGET,\\sdssdp23\dp23.b\data\csv\phCSV\target\0-37-
718022\,JohnDoe,targ39
    DR2,TARGET,\\sdssdp23\dp23.b\data\csv\phCSV\target\0-82-
1113106\,JohnDoe,targ40
```


SDSS Load Monitor - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Media
Links
File Upload
sqlLoader v4_8_7
| Home | Help | Servers | Stats | Error | Warning | New Task | Upload |
| Active Tasks | Finished | Destinations | Killed | Null Task | All Tasks |
Browse...
Submit
Reset
Internet

Cut and paste the list here:

```
#
#this is a comment line
#
TEST,BEST,\\sxatlas\staging\loadtest\phCSV\best\1-43-278-938\,user,comment...
TEST,TARGET,\\sxatlas\staging\loadtest\phCSV\best\1-43-278-938\,user,no comment...
TEST,FINISH,\\,user,finish it up
#
```

Or upload it as text file

**Figure 9. Load Monitor File Upload page to submit multiple tasks in one shot. Tasks can be entered in the upload window or a file upload can be specified.**

### 7.1.4 Killing a Task

A task can be killed by clicking on the last column of the task display in the tasks table. The user is prompted for confirmation. The loader cleans up when a task is killed, but some files and especially the temporary task DBs created will not be deleted until the same task (with the same parameters but of course a different taskID) is run again. This is intentional because recreating a DB is a time-consuming process and the assumption is that in the vast majority of cases, a task that is killed will be rerun at a later date, since that data must be loaded into the database eventually. Of course, the task DB can always be manually deleted in EM after the task is killed.

### 7.1.5 Monitoring the Load

Selecting the Active Tasks or All Tasks links in the Load Monitor shows you the tasks that are currently running. The color-coding for the task status is shown below the task table. For each task, the taskid, the stepid and the phase number are shown, along with the name of the task and step that is currently being executed. The task display is updated once every minute.

For each task, you can select the Steps, Files or Log links to look at the steps, files and phases logged (completed) for that task.

The PRELOAD step of a loading task usually takes the longest time, as the CSV files are loaded into the load-DB in this step. The largest of the CSV files for each run - the PhotoObj*.csv files - each will take 10-15 mins each to load, and the preload step for one imaging chunk can take more than an hour to complete. You can monitor the progress of the preload step by selecting the Files display for that task.

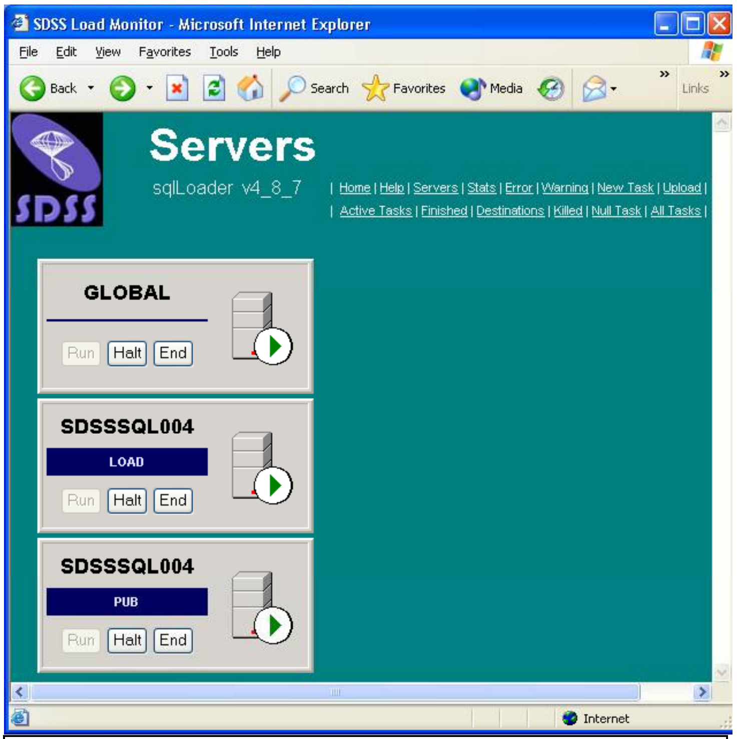

SDSS Load Monitor - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Links
SDSS
Servers
sqlLoader v4_8_7
| Home | Help | Servers | Stats | Error | Warning | New Task | Upload |
| Active Tasks | Finished | Destinations | Killed | Null Task | All Tasks |
GLOBAL
Run Halt End
SDSSSQL004
LOAD
Run Halt End
SDSSSQL004
PUB
Run Halt End
Internet

**Figure 10. Load Monitor Servers page. The loader-admin can monitor, stop and restart servers from this page.**

## 7.2 Storage Requirements for Loading

In order for the loading to proceed smoothly and expediently, a certain minimum amount of disk storage is recommended. As Fig. 11 shows, in addition to the loadserver, there are 3 copies of each dataset that are kept spinning on disk – the 2 production copies (one live and one warm spare) and the legacy copy that contains all the data served up to date. We maintain a pingpong configuration for the loading, i.e. we alternate between 2 servers so that while one of them is being loaded, the other is pressed into service as the live production server. Each of these servers must have twice the amount of storage required to store one copy of the archive, in order to store backups of the task DBs that are created during the loading process. It is generally also a good idea to have this amount of spare storage if it is necessary to recover the DB from a backup (recovery

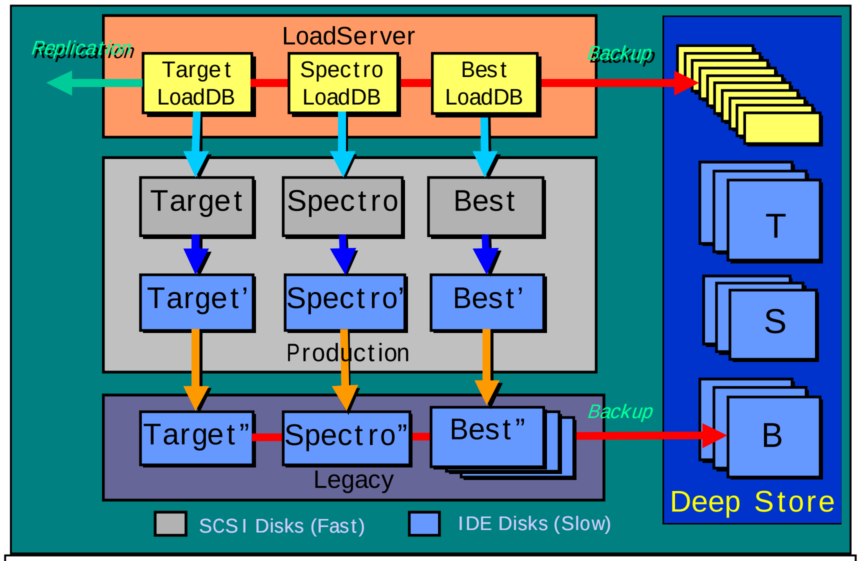


**Figure 11. Copies of the SDSS Archive that are necessary for the loading and production configuration, including a warm spare, a legacy backup and offline tape (deep store) backup.**

temporarily requires twice the amount of space). This means that in addition to the space required for the legacy copies, disk storage equal to at least 5 times the size of the archive must be available. Our experience has shown that even this is not sometimes enough if we allow for disk fragmentation, disk errors etc., and actually we recommend having 6 times the space required for one instance of the archive to be available. This may sound like a lot, but when dealing with Terabyte datasets, one wants to avoid having to reload the databases or having to copy them from one server to another as much as possible. Shuffling data between servers in order to make room is just not a good option.

## 8 Future Work

There are three main areas where we intend to concentrate future development on the sqlLoader product:

- Making the loading completely automated.
- Making it scalable to petabyte-scale archives.
- Generalizing the sqlLoader to any schema.
- Extending the pipeline to other DBMS environments beyond SQL Server.

Both are challenging objectives, but the second goal definitely entails a lot more work. We describe in more detail the steps that we will have to undertake for each of these objectives.

### 8.1 Making sqlLoader Fully Automated

The following are some of the situations in which manual intervention is still required for sqlLoader operation, so we are working on including them in the automated machinery.

1. **Setting up user privileges** – The first time that a newly published database is accessed, the user privileges are often not set correctly for users to access the various tables, stored procedures and functions. The problem has to do with mapping the preexisting login for the web user to the username defined in the publish database. This is referred to as the “orphaned user” problem, i.e. database

users are orphaned because they do not match up with the SQL server logins defined for that database server.
2. Incremental loading – the ability to checkpoint a published database and continue loadind data incrementally above it is crucial for efficient archive operations and quick availability of the data to scientists.
3. Backing out and reloading data – there will inevitably be errors in the input data, and hence the need will arise occasionally to back out a defective chunk of data and reload it.
4. Error handling – errors encountered during loading currently require manual intervention for the most part.

## 8.2 Scalability – Partitioning the Loaded Data

At JHU, we are developing a partitioning algorithm to horizontally partition the SDSS database tables, as part of a NASA AISRP-funded project to develop a high-speed parallel data access prototype for the NVO. The motivation for partitioning data springs mainly from the following performance considerations:

### 8.2.1 Speeding up spatial query execution

We would like to speed up query execution for spatial queries that are frequently submitted in astronomical data mining applications. By horizontally partitioning the data, we aim to achieve the following results:

- Queries looking at different parts of the sky are distributed among servers. This would speed up the bulk of normal astronomy queries by providing a type of load balancing.
- Queries covering wide areas are executed in parallel by different servers. Sequential scans fall naturally in this range.
- Proximity searches (neighborhood queries) are "isolated" and processed in parallel. Searches involving gravitational lenses and galaxy clusters are in this group.

### 8.2.2 Speeding up cross-matching queries

The worldwide federation of archives promised by the international Virtual Observatory effort will give rise to virtual data grids that will enable VO users to submit cross-matching requests over substantial spatial subsets of the individual archives through distributed VO queries. As part of another AISRP-funded project, we have already built a prototype distributed query service, SkyQuery (www.skyquery.net), that performs cross-matches through probabilistic joins between geographically distributed archives. However, the cross-match intersection area that can be specified with SkyQuery is at present practically limited due to performance constraints to a few square degrees.

In spite of the ingenious recursive cross-matching algorithm that SkyQuery employs, it is prohibitively expensive to perform a cross-match over a substantial fraction of the area covered by the largest archives. Speeding up cross-match requests from other NVO data nodes is therefore the second big motivation behind spatial partitioning of the data in large archives like SDSS.

#### 8.2.2.1 Partitioning Strategy

Partitioning works well if the following two requirements are met.

1. Data is partitioned uniformly and symmetrically across the different servers.
2. Tables in the database are naturally divisible into similar partitions where all rows accessed by the query are on the same server.

Our partitioning strategy is therefore the following:

- Distribute data homogenously among servers. This implies two things:
    - Each server has roughly the same amount of objects.
    - Objects inside a given server are spatially related. We define *buffers* to build neighborhoods, and *margins* for "fuzziness" around each zone in which objects from other servers are duplicated, so as to avoid queries that need data from more than one server.

  This distribution balances the workload among servers so that queries are redirected to the server holding the data.
- Define zones inside each server dynamically. Zones are defined according to some search radius to solve specific problems, such as finding galaxy clusters, gravitational lenses, etc. This facilitates cross-match queries from other NVO data nodes.

Once we have developed and tested a partitioning system with our SDSS data, the main task will be to incorporate this algorithm into the data import pipeline. The most logical place to add this functionality in the pipeline will be in the final Finish stage, before the database indices are created. A secondary task will be translating this logic to standard SQL or other SQL dialects if necessary.

## 8.3 Schema Independence

Extending the current data import pipeline to non-SDSS schemas will require a significant amount of work, but we do not see any major technical hurdles in achieving this. The details of the schema can be parameterized by reading the DDL from files that are set up locally at each installation. The schema currently is already localized to a schema subdirectory within the sqlLoader files.

Coupling to the SDSS schema exists currently in the following parts of the loader pipeline:

- Schema creation in the task (temporary) DBs – each of the temporary DBs has a replica of the full SDSS schema in them.
- Schema creation in the publish DBs – this is the place where all the data ends up.
- Data validator – some of the validating tests are SDSS-specific and tied to the existing schema.
- Finish/merge step – where database indices, pre-computed joins and ancillary tables and views are created.

For the schema creation, extension to other schemas should be straightforward because these files are simply executed as SQL scripts each time the databases are created. Making the validator and finish step generic will require the changes discussed below.

### 8.3.1 Data Validator

The uniqueness tests in the Validator are already set up so that individual tests call a primitive stored procedure spGenericTest to actually perform the test. However, the foreign key tests are hard-coded with the name of each foreign-key that needs to be tested. The list of foreign keys to be tested needs to be driven by a single indexMap table, which contains the list of all primary and foreign keys in the database. Then the same table will drive the creation of indices.

There are also HTM-ID tests and parent-child link consistency tests that are specific to the SDSS schema tables. These tests will be set up in such a way that the table names and other parameters can be obtained from another table.

We plan to add a Validator Setup Script that will allow the archive administrator to select, add or remove different kinds of validation tests and adjust the parameters for each test as applicable. This script will provide an easy interface to customize the validation for each schema and installation site. An ASCII script interface is chosen rather than a GUI setup tool so as to make this a versatile and platform-independent facility.

### 8.3.2 Finish/Merge Step

The Finish/Merge step is currently a stored procedure that creates all the database indices, creates tables that are pre-computed joins of other tables, and ties up all the loose ends from the previous loading steps. The index creation will also be driven from the IndexMap table created in the publish DB. By another stored procedure that manages all the indices. To make the index creation customizable to other schemas, this procedure will have to be modified so that it fills the IndexMap table from a locally built file that contains all the index names and parameters.

As in the case of the Validator, we will provide a setup script for the Finish step that will enable archive administrators to specify:

- the file from which to read in the index parameters
- the names of scripts or procedures to execute for remaining Finish tasks. These will have to be written by the archive administrator to suit the schema for that archive.

Thus there will be three tasks related to schema independence:

1. Parameterize the schema creation procedures – load the schema from configurable files.
2. Rewrite the Validator so that the various tests are configured in a Validator Setup Script.
3. Rewrite the Finish so that indices and final tasks are configured in a Finish Setup Script.

## 8.4 DBMS Independence

This will be a very challenging task, therefore we have decided to initially focus our efforts on the IBM DB2 DBMS as the primary test of our ideas. If we are successful in deploying our data import pipeline on DB2, we will have made sufficient changes to make migration to other systems such as Oracle an incremental effort beyond DB2. Our choice of DB2 is motivated by its multi-platform support, its reasonable licensing terms and (not unrelated) popularity within the NVO community.

There are two main aspects in which we will need to change the current pipeline in order to migrate it to a different DBMS:

1. Administrative functions – the administrative functions like scheduling tasks, setting up user privileges will be different on DB2.
2. Dialect of SQL – we will have to translate the scripts and stored procedures currently written in T-SQL (Transact-SQL, the SQL Server dialect of standard SQL) to DB2's dialect, which is simply called DB2 SQL dialect. The latter is more powerful than T-SQL, so we do not anticipate running into major roadblocks (i.e., things that *cannot* be done in DB2 SQL dialect).

There are several ways in which the current pipeline is tied to the SQL Server DBMS and to T-SQL (the SQL Server SQL dialect). These are discussed below in increasing order of difficulty of migrating them to a different DBMS with a different dialect of SQL. Each of them will generate a well-defined task for the project plan.

### 8.4.1 Translation of Stored Procedures

T-SQL stored procedures will be translated into the DB2 SQL dialect equivalent. We will decide whether it is better to translate them to Java stored procedures or SQLJ which DB2 supports. The main reason for not having Java stored procedures for our current SQL Server is the loss of speed with the JDBC interface.

The translation will be more of a tedious than a difficult task, especially if done manually. We plan to write a translator (a Perl or VB script) to do it.

### 8.4.2 SQL Server Agent job scheduling

The loader workflow is maintained by defining two jobs for the SQL Server Agent process. Both jobs are set up to run once a minute and pick up the next batch of work when they run. The creation of these jobs is scripted in the loader stored procedures. These commands will have to be translated to the equivalent commands on other DBMSs.

### 8.4.3 HTM SQL Interface

The Hierarchical Triangular Mesh (HTM) is a spatial indexing scheme developed at JHU to partition the sky recursively into spherical triangles with unique HTM IDs and storing the IDs in a quad-tree [Kunszt00]. The HTM is available as an open-source library (http://www.sdss.jhu.edu/htm/) in C++ and Java (and C# soon). We have incorporated the HTM into SQL Server by writing a SQL interface to the HTM functions.

There are a handful of SQL functions that provide this interface between the database and the HTM software. Some of these are extended stored procedures that interface to primitives in the HTM DLL. These glue functions will have to be ported to the new dialect of SQL.

### 8.4.4 The Zones Algorithm

The Zones algorithm [Gray02] is an even faster alternative to HTM spatial lookups that uses SQL operators and a subdivision of the sky into declination *zones* to perform a spatial search. The HTM↔SQL interface is a lot slower than using direct SQL operators because every HTM call is an expensive call to an external function. Although this will change once C# is integrated into SQL Server with the Yukon release (so that the HTM DLL can be moved into the database), the Zones algorithm still provides a quick and convenient (low overhead, no need to implement HTM) way of subdividing the spatial area, and is very useful for horizontally partitioning the database tables.

### 8.4.5 Security and User Administration

The loader pipeline sets up the Web access privileges to the published databases also as part of the schema creation. This includes creating and setting up the privilege level (handled through user roles in SQL Server) for the Web access user, setting accessibility levels for individual tables, stored procedures and functions. This is currently done with T-SQL statements, and we expect that we will be able to translate these to DB2 SQL dialect quit easily. It remains to be seen whether privileges for the “loadagent” user, which our pipeline uses to perform administrator level tasks currently, can be set in the same way on DB2 and other systems.

### 8.4.6 DTS packages

The Data Transformation Services subsystem in SQL Server allows several types of data operations. This will probably be the hardest subtask within this task because there may or may not be an equivalent tool on other DBMSs. Currently the DTS tasks are set up interactively using SQL Server’s DTS wizard, and the package is saved to disk once the setup is completed. The saved package is then programmatically invoked from within the loader pipeline code. In all likelihood, DTS packages will have to be rewritten as SQL, Jscript or Perl (for UNIX/Linux) scripts on other systems like DB2.

# 9 Appendices

## 9.1 sqlLoader Workflow

There are three distinct stages in the loader workflow:

1. Load-Validate-Publish – this stage includes checking the input CSV files, loading them into temporary task DBs, validating the data in the task DBs and publishing the data to the publish DB.
2. Merge – Merging of the various data streams (imaging, spectro, tiling) in the publish DB and creating the indices on the main tables.
3. Finish – running the final tests, creating the derived tables and precomputed joins, and the corresponding indices.

Stages 2 and 3 must be run in sequential at the moment, whereas stage 1 can be run in parallel on a cluster of load servers. The workflows for the different stages are described in detail below.

## 9.2 LOAD Workflow for a TASK Database

1. **Export**
    1. **Start Step**
    2. **Verify that task does not exist**
    3. **Create new entry in loadadmin.Task**
    4. **End Step**
2. **Check**
    1. **Start Step**
    2. **Check for the existence of root directory**
    3. **Compare root path to export-type**
        1. Verify existence of csv_ready
        2. Check each CSV file vs csv_ready
        3. Insert each CSV filename into loadadmin.Files
    4. **Check each subdirectory (Run or Plate)**
        1. Verify existence of csv_ready file
        2. Check each CSV file vs csv_ready
        3. Insert each CSV filename into loadadmin.Files
        4. Check for Zoom or Gif subdirectories
        5. Count the number of files
        6. Insert subdirectory name, if Zoom into loadadmin.Files
    5. **End Step**
3. **Build**
    1. **Start step**
    2. **Create TaskDB**
        1. Drop database if already exists
        2. Delete DB files, if they exist
        3. Create the database
        4. Configure DB options

        5. Schema
            1. Execute 'dataConstants.sql'
            2. Execute 'constantSupport.sql'
            3. Execute 'metadataTables.sql'
            4. Execute 'photoTables.sql'
            5. Execute 'spectroTables.sql'
            6. Execute 'views.sql'
            7. Execute 'spHTM.sql'
            8. Execute 'nearFunctions.sql'
            9. Execute 'zoomTables.sql'
            10. Execute 'webSupport.sql'
            11. Execute 'boundary.sql'
            12. Execute 'myTimeX.sql'
            13. Execute 'spSetValues.sql'
            14. Execute 'spValidate.sql'
            15. Execute 'spManageIndices.sql'
            16. Execute 'spBackup.sql'
            17. Execute 'spPublish.sql'
            18. Execute 'spFinish.sql'
            19. Execute 'spGrantAccess.sql'
    3. **End Step**

4. **Preload**
    1. **Start step**
    2. **spFileLoop**
        1. Load each CSV file
        2. Load each Zoom directory
        3. Load each plate
        4. Load each tileRun
        5. Test if all files loaded
    3. **Test row count against total number of lines listed in csv_ready file(s)**
    4. **Set Values**
        1. Get Task type
        2. Set loadVersion
            1. for PlateX
            2. for HoleObj
            3. for Tile
            4. for TileBoundary
            5. for TiledTarget
            6. for TileInfo
            7. for TileRegion
            8. for Chunk
            9. for Segment
            10. for Field
            11. for Target
            12. for TargetInfo
        3. SetValues (Plates)
            1. Update SpecObjAll(objType, loadVersion)

        4. Set Values (Photo)
            1. Set simplified mags (u,g,r,i,z, Err_*, dered_*, loadVersion)
            2. Remove ObjMask rows in hole fields
            3. Exec spComputeFrameHTM
            4. Exec spTargetInfoTargetObjId
            5. Update Mask(cx,cy,cz)
            6. Update Mask(htmId)
            7. -- Compute Boundaries  should move to Finish
    5. **Check that each file/directory in Files is DONE**
    6. **Check that the number of lines for each file adds up**
    7. **End Step**
5. **Validate**
    1. **Start Step**
    2. **Validate Photo**
        1. Test unique keys
            1. Chunk(chunkId)
            2. Segment(segmentID)
            3. StripeDefs(stripe)
            4. Field(FieldID)
            5. FieldProfile(fieldID, bin, band)
            6. Frame(FieldID,Zoom)
            7. PhotoObj(objID)- this takes pretty long (20mins-half hour ballpark)!!
            8. PhotoProfile(objID, bin, band) - this takes even longer (nearly an hour or more)!!
            9. PhotoZ(objID,rank, pid)
            10. First(objID)
            11. Rosat(objID)
            12. USNO(objID)
            13. Mask(maskID)
            14. ObjMask(objID)
            15. Target (targetID)
            16. TargetInfo(skyVersion,targetID)
            17. TargetInfo(targetObjID) (not valid for South)
            18. TargetParam(targetVersion,paramName)
        2. Test Foreign keys
            1. Chunk(stripe)  -> StripeDefs(stripe)
            2. Segment(stripe)  -> StripeDefs(stripe)
            3. Segment(ChunkID) ->  Chunk(chunkID)
            4. Field(segmentID) ->  Segment(segmentID)
            5. Frame(fieldID)  ->  Field(fieldID)
            6. FieldProfile(fieldID) ->  Field(fieldID)
            7. PhotoObj(fieldID)  ->  Field(fieldID)
            8. PhotoProfile(ObjID) ->  PhotoObj(ObjID)
            9. PhotoZ(ObjID)  ->  PhotoObj(ObjID)
            10. ObjMask(ObjID) -> PhtoObj(ObjID)
            11. First(ObjID)  ->  PhotoObj(ObjID)
            12. Rosat(ObjID)  ->  PhotoObj(ObjID)
            13. USNO(ObjID)  -> PhotoObj(ObjID)

      14. TargetInfo(targetID) -> Target(targetID)
   3. Test cardinalities, also against the files table
      1. Mask(run, rerun, camcol, field)  ->  Field(run, rerun, camcol, field)
      2. Segment(nFields) = count(Fields) on segmentID
      3. Field(nObjects)  = count(PhotoObj) on fieldID
      4. count(Field) * 7 = count(Frame)    on fieldID  (tests zoom levels)
      5. PhotoObj(nProf_ugriz) = count(PhotoProfile)   on objID
   4. Test HTM id's
      1. Frame(HTM)
      2. Mask(HTM)
      3. PhotoObj(HTM)
   5. Test parents (nChild consistency)
   6. Write summary
3. **Validate Plates**
   1. Test unique keys
      1. PlateX(plateID)
      2. SpecObjAll(specObjID)
      3. ELRedshift(ELRedshiftID)
      4. SpecLineAll(SpecLineID)
      5. SpecLineIndex(SpecLineIndexID)
      6. XCRedshift(XCRedshiftID)
      7. HoleObj(holeID)
   2. Test Foreign keys
      1. SpecObjAll(plateID)   -> PlateX(plateID)
      2. ElRedshift(specObjID)  -> SpecObjAll(specObjID)
      3. SpecLineAll(specObjID) -> SpecObjAll(specObjID)
      4. SpecLineIndex(specObjID) -> SpecObjAll(specObjID)
      5. XCredshift(specObjID)  -> SpecObjAll(specObjID)
      6. HoleObj(plateID)  -> PlateX(plateID)
   3. Test HTM id's
      1. SpecObjAll(HTM)
4. **Validate Tiles**
   1. Test unique keys
      1. Tile(tile)
      2. TileBoundary(tileBoundID)
      3. TiledTarget(targetID,tile)
      4. TileInfo(tileRun, tid)
      5. TileNotes(tileNoteID)
      6. TileRegion(tileRun)
   2. Test Foreign keys
      1. Tile(tileRun)  ->  TileRegion(tileRun)
      2. TileBoundary(tileRun) -> TileRegion(tileRun)
      3. TileBoundary(Stripe) -> StripeDefs(stripe)
      4. TileInfo(tileRun)  ->  TileRegion(tileRun)
      5. TileNotes(tileRun)  -> TileRegion(tileRun)
      6. TiledTarget(tile)  ->  Tile(tile)
5. **Write summary**
6. **End Step**

6. **Backup**
    1. **Begin Step**
    2. **Backup TaskDB to brick for archival storage**
    3. **End Step**
7. **Detach**
    1. **Begin Step**
    2. **Detach TaskDB**
    3. **End Step**

## 9.3 PUB Workflow for a TASK Database

1. **Publish**
    1. **Start Step**
    2. **Start Step in PubDB**
    3. **Attach TaskDB to publish server**
    4. **Insert/transform each table into destination**
        1. Publish Photo
            1. publish Chunk
            2. publish Segment
            3. publish Field
            4. publish Frame
            5. publish FieldProfile
            6. publish PhotoObj
            7. publish PhotoProfile
            8. publish PhotoZ
            9. publish First
            10. publish Rosat
            11. publish USNO
            12. publish Mask
            13. publish ObjMask
            14. publish Target
            15. publish TargetInfo
            16. publish TargetParam
        2. Publish Plates
            1. publish PlateX
            2. publish specObjAll
            3. publish Elredshift
            4. publish specLineAll
            5. publish specLineIndex
            6. publish Xcredshift
            7. publish HoleObj
        3. Tiling Publish
            1. publish Tile
            2. publish TileBoundary
            3. publish TiledTarget

      4. publish TileInfo
      5. publish TileNotes
      6. publish TileRegion
   5. **Detach TaskDB**
   6. **Update LoadHistory**
   7. **End Step in PubDB**
   8. **End Step**
2. **Cleanup**
   1. **Begin Step**
   2. **Delete TaskDB**
   3. **End Step**

## 9.4 Workflow for a PUB Database

1. **Build PubDB**
   1. **Start Step**
   2. **Create PubDB**
   3. **Build schema (See TaskDB.Build.Schema)**
   4. **Load metadata tables**
      1. DBColumns
      2. DBObjects
      3. DBViewcols
      4. Glossary??
   5. **End Step**
2. **Publish**
   1. **See TaskDB.Publishâ€¦**
3. **Finish**
   1. **Begin Step in PubDB**
   2. **Get DB parameters**
   3. **Drop all indices**
   4. **Build all indices**
      1. Build Primary keys
         1. Chunk(chunkId)
         2. Segment(segmentID)
         3. StripeDefs(stripe)
         4. Field(FieldID)
         5. Frame(FieldID,Zoom)
         6. FieldProfile(fieldID, bin, band)
         7. PhotoObj(objID
         8. PhotoProfile(objID, bin, band)
         9. PhotoZ(objID,rank, pid)
         10. First(objID)
         11. Rosat(objID)
         12. USNO(objID)

13. Mask(maskID)
14. ObjMask(objID)
15. Target (targetID)
16. TargetInfo(skyVersion,targetID)
17. TargetParam(targetVersion, paramName)
18. PlateX(plateID)
19. SpecObjAll(specObjID)
20. ELRedshift(ELRedshiftID)
21. SpecLineIndex(SpecLineIndexID)
22. SpecLineAll(SpecLineID)
23. XCRedshift(XCRedshiftID)
24. HoleObj(holeID)
25. Tile(tile)
26. TileBoundary(tileBoundID)
27. TileInfo(tileRun, tid)
28. TileNotes(tileNoteID)
29. TileRegion(tileRun)
30. TiledTarget(targetID) *** no key known yet
31. Globe(globeId)
32. Frame(fieldID,zoom)
33. Glossary(key)
34. History(version)
35. DbObjects(name)
36. DBColumns(tableName,name)
37. DBViewCols(viewName,name)
38. DataConstants(field,name)
39. Diagnostics(name)
40. SDSSConstants(name)
41. SiteConstants(name)
42. SiteDiagnostics(name)
43. LoadHistory(loadVersion, tStart)
44. ProfileDefs(bin)
45. Area(areaID)
46. if Build all: Neighbors(objID, NeighborObjID)
47. if Build all: Zone(ZoneID, ra, objID)
48. if Build all: TiBound2tsChunk(tileBoundID)
49. if Build all: Sector(SectorID)
50. if Build all: Best2Sector(bestObjID)
51. if Build all: Target2Sector(targetID)
52. if Build all: Sector2Tile(SectorID, tile)

2. Build the other indices
    1. ELRedShift(specObjID,elRedShiftID)
    2. Field(field,camcol,run,rerun)
    3. Frame(field,camcol,run,zoom,rerun)
    4. Frame(htmID,zoom,cx,cy,cz,a,b,c,d,e,f,node,incl)
    5. PhotoObj(mode,cy,cx,cz,htmID,type,flags,status,ra,dec,u,g,r,i,z,rho)
    6. PhotoObj(htmID,cx,cy,cz,type,mode,flags,status,ra,dec,u,g,r,i,z,rho)

7. PhotoObj(field,run,rerun,camcol,type,mode,flags,rowc,colc,ra,dec,u,g,r,i,z)
8. PhotoObj(fieldID,objID,ra,dec,r,type,status,flags)
9. PhotoObj(SpecObjID,cx,cy,cz,mode,type,flags,status,ra,dec,u,g,r,i,z,rho)
10. PhotoObj(parentID,cx,cy,cz,mode,type,flags,status,ra,dec,u,g,r,i,z,rho)
11. PhotoObj(cx,cy,cz,htmID,mode,type,flags,status,ra,dec,u,g,r,i,z,rho)
12. PhotoObj(run,mode,type,status,flags,u,g,r,i,z,Err_u,Err_g,Err_r,Err_i,Err_z)
13. PhotoObj(run,camcol,rerun,type,mode,status,flags,ra,dec,fieldID,field,u,g,r,i,z)
14. PhotoObj(run,camcol,field,mode,parentID,q_r,q_g,u_r,u_g,isoA_r, isoB_r,fiberMag_u, fiberMag_g,fiberMag_r,fiberMag_i,fiberMag_z)
15. unique SpecLineAll(specobjID,specLineID)
16. unique SpecLineIndex(specobjID,speclineindexID)
17. unique SpecObjAll(TargetObjID,objType,objTypeName,sciencePrimary, specClass, htmID,ra,dec,fiberMag_u,fiberMag_g,fiberMag_r,fiberMag_i, fiberMag_z)
18. SpecObjAll(BestObjID,objType,objTypeName,sciencePrimary,specClass,htmID, ra,dec, fiberMag_u,fiberMag_g,fiberMag_r,fiberMag_i,fiberMag_z)
19. SpecObjAll(specClass,zStatus,zWarning,z,sciencePrimary,primTarget, secTarget, plateId,bestObjID,targetObjId,htmID,ra,dec)
20. unique XCRedshift(specObjID,xcRedShiftID)
21. DataConstants(value)

3. Build foreign keys
    1. Chunk(stripe) -> StripeDefs(stripe)
    2. Segment(stripe) -> StripeDefs(stripe)
    3. Segment(ChunkID) -> Chunk(chunkID)
    4. Field(segmentID) -> Segment(segmentID)
    5. Frame(fieldID) -> Field(fieldID)
    6. FieldProfile(fieldID) -> Field(fieldID)
    7. PhotoObj(fieldID) -> Field(fieldID)
    8. PhotoZ(ObjID) -> PhotoObj(ObjID)
    9. PhotoProfile(ObjID) -> PhotoObj(ObjID)
    10. ObjMask(ObjID) -> PhtoObj(ObjID)
    11. First(ObjID) -> PhotoObj(ObjID)
    12. Rosat(ObjID) -> PhotoObj(ObjID)
    13. USNO(ObjID) -> PhotoObj(ObjID)
    14. TargetInfo(targetID) -> Target(targetID)
    15. SpecObjAll(plateID) -> PlateX(plateID)
    16. ElRedshift(specObjID) -> SpecObjAll(specObjID)
    17. SpecLineAll(specObjID) -> SpecObjAll(specObjID)
    18. SpecLineIndex(specObjID) -> SpecObjAll(specObjID)
    19. XCredshift(specObjID) -> SpecObjAll(specObjID)

            20. HoleObj(plateID)  -> PlateX(plateID)
            21. Tile(tileRun)  ->  TileRegion(tileRun)
            22. TileBoundary(tileRun) -> TileRegion(tileRun)
            23. TileBoundary(Stripe) -> StripeDefs(stripe)
            24. TileInfo(tileRun)  ->  TileRegion(tileRun)
            25. TileNotes(tileRun)  -> TileRegion(tileRun)
            26. TiledTarget(tile)  ->  Tile(tile)
            27. PlateX(tile)   ->  Tile(tile)
            28. if Build all: Neighbors(objId)-> PhotoObh(objId)
            29. if Build all: Target(BestObj) will be computed at finish
            30. if Build all: TileBoundary(tileBoundID) -> TiBound2tsChunk (tileBoundID) ** TiBound2tsChunk is empty
            31. if Build all: Target2Sector(sectorID) ->  Sector(sectorID)
            32. if Build all: Target2Sector(targetID) -> Target(TargetID)
            33. if Build all: TiBound2tsChunk (tileBoundID)-> TileBoundary(tileBoundID)
            34. if Build all: TiBound2tsChunk (chunkID)-> Chunk(chunkID)
            35. if Build all: Best2Sector(SectorID) -> Sector(SectorID)
            36. if Build all: Sector2Tile(SectorID) ->  Sector(SectorID)
            37. if Build all: Sector2Tile(tile)  ->  Tile(tile)

5. **If BEST-PUB: Finish Plates**
    1. Get TARGET-PUB name
    2. Drop Target* indices
    3. Copy Target into BEST-PUB (incremental)
    4. Copy TargetInfo into BEST-PUB (incremental)
    5. Copy TargetParam into BEST-PUB (incremental)
    6. Mark TiledTarget duplicates
    7. Set Target(bestObjId) with nearest PhotoObj in BEST-PUB
    8. Rebuild Target* indices
    9. Set SpecObjAll(targetObjid) to TargetInfo(targetObjid)
    10. Set SpecObjAll(sciencePrimary) flag
    11. Run SpectroPhoto matchup
        1. Create TAB variable using fGetNearestObjIdEq
        2. Update distance
        3. Delete orphans (no match)
        4. Update SpecObjAll(bestObjid)
        5. Update PhotoObj(specObjId)
    12. Do Sector computations
    13. Build Sector indices
    14. Compute sampling probabilities for Sectors
6. **Finish Photo**
    1. Compute Neighbors
        1. Set radius according to type (30 arcsec if BEST, 3 arcsec else)
        2. Build zone table (30 arcsecond zones)
        3. Build Primary Key for Zone
        4. Truncate Neighbors table
        5. Compute -1, 0, 1 zone
        6. Add in neighbor mirrors

            7. Build Primary Key index on Neighbors(objid, neighborObjid)
        2. Compute Chain table
    7. **Update Statistics: *** commented out for now**
    8. **End Step in PubDB**

## 9.5 Workflow to Synchronize the PUB Databases

1. **FINISH - run on each PubDB**
    1. **Begin Step in PubDB**
    2. **Drop all indices**
    3. **Build (almost) all indices**
    4. **Compute Neighbors (instead of FinishPhoto)**
    5. **Compute Chains**
    6. **Compute inMask stuff**
    7. **End Step in PubDB**
2. **LINK RUNS**
    1. **Compute RUNS-BEST cross-links into RUNS**
        1. link the heads of chains together
3. **LINK TARGET**
    1. **Compute BEST-TARGET cross-links into BEST**
        1. link the heads of chains together
4. **SYNC BEST**
    1. **Update from Target**
        1. Copy Target
        2. Copy TargetInfo
        3. Copy TargetParam
    2. **Update SpecObjAll**
    3. **SpectroPhoto matchup**
    4. **Sector computations**
        1. Compute sector boundaries
    5. **Build remaining indices**
        1. Build Target* indices
        2. Build Sector* indices
    6. **Compute Sampling Probabilities**
5. **REPARTITION**

Should this not be done at publish already?

6. **REORG**
    1. **Run spReorg**
    2. **Update statistics**

   3. **Run Checksum, insert into SiteDiagnostics**

7. **TABLES THAT WE ARE IGNORING NOW**
   1. Area -- needed for the boundary stuff
   2. Edge
   3. Rmatrix
   4. Chain  need to create it

We will need to deal with MOSAIC stuff